

# Optimising Neural Network Performance for Digit Classification with Random Search, Bayesian Optimization, and Genetic Algorithms

May 9, 2024

## Project Introduction:

This project aims to optimize the performance of a neural network model for classifying handwritten digits from the MNIST dataset. The MNIST dataset consists of 28x28 pixel grayscale images of handwritten digits (0 to 9) and is a popular benchmark dataset for evaluating machine learning algorithms. The goal is to find the best set of hyperparameters for the neural network model that maximizes its accuracy in digit classification.

## Dataset Description:

The MNIST dataset consists of 60,000 training images and 10,000 test images. Each image is a 28x28 pixel grayscale image of a handwritten digit. The task is to classify each image into one of the ten possible classes (digits 0 through 9).

**Optimization Algorithms:** Three different optimization algorithms will be used to find the best set of hyperparameters for the neural network model:

**Random Search:** Randomly selects hyperparameter combinations from a predefined search space. Iteratively evaluates each combination to find the one that yields the best performance.

**Bayesian Optimization:** Uses a probabilistic model to select the most promising hyperparameter combinations. Updates the model based on the performance of previously evaluated combinations.

**Genetic Algorithm:** Mimics the process of natural selection to find the best hyperparameter combination. Maintains a population of candidate solutions and evolves them over successive generations.

**Neural Network Model:** The neural network model consists of three fully connected layers:

Input layer with 784 neurons (corresponding to the flattened 28x28 pixel images). Two hidden layers with 512 neurons each, using the ReLU activation function. Output layer with 10 neurons (one for each digit), using the softmax activation function. The optimization algorithms will search for the best combination of hyperparameters, including:

Optimizer (Adam, RMSprop, or SGD) Number of epochs (5, 15, or 25) Batch size (128 or 256)

**Objective:** The objective is to find the hyperparameters that maximize the accuracy of the neural network model in classifying the digits in the MNIST dataset. By comparing the performance of the three optimization algorithms, we aim to determine which algorithm is most effective in finding the optimal hyperparameters for the neural network model.

```
[ ]: !pip install bayesian-optimization[sklearn]
!pip install bayesian-optimization
!pip install scikeras
import random
import numpy as np
from bayes_opt import BayesianOptimization
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import SGD
from scikeras.wrappers import KerasClassifier
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from sklearn.metrics import accuracy_score
from tensorflow import keras
from tensorflow.keras.layers import Activation
from tensorflow.keras.utils import to_categorical
from scikeras.wrappers import KerasClassifier
```

Random Search Algorithm:

```
[ ]: num_classes = 10

# Load the MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Reshape the data
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)

# Convert the data type to float32
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

# Normalize the data
X_train /= 255.0
X_test /= 255.0

# Convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

# Define the model
model = Sequential()
model.add(Dense(512, input_shape=(784,)))
```

```

model.add(Activation('relu'))
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dense(num_classes))
model.add(Activation('softmax'))

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam',
              metrics=['accuracy'])
# Wrap the Keras model with KerasClassifier
model = KerasClassifier(model=model)

# Define the hyperparameter space
hyperparameter_space = {
    "optimizer": ["adam", "rmsprop", "sgd"],
    "epochs": [5, 15, 25],
    "batch_size": [128, 256]
}

# Create the RandomizedSearchCV object
random_search_cv = RandomizedSearchCV(
    estimator=model,
    param_distributions=hyperparameter_space,
    n_iter=1,
    random_state=1
)

# Fit the RandomizedSearchCV object to the data
random_search_cv.fit(X_train, y_train)

# Print the best hyperparameters found by the search
print("Best hyperparameters:")
print(random_search_cv.best_params_)

# Train the best model with the best hyperparameters
best_model = random_search_cv.best_estimator_

# Evaluate the best model on the test data
test_loss, test_accuracy = best_model.model.evaluate(X_test, y_test, verbose=0)

# Print the test results
print("Test loss:", test_loss)
print("Test accuracy:", test_accuracy)

# Get predictions from the best model
predictions = best_model.predict(X_test)

```

```

# Calculate the accuracy of the best model
accuracy = np.sum(np.argmax(predictions, axis=1) == np.argmax(y_test, axis=1)) /
    ↪ len(y_test)

# Print the accuracy of the best model
print("Best model accuracy:", accuracy)

# Print the confusion matrix of the best model
from sklearn.metrics import confusion_matrix
confusion_matrix = confusion_matrix(np.argmax(y_test, axis=1), np.
    ↪ argmax(predictions, axis=1))
print("Confusion matrix:")
print(confusion_matrix)

```

Results:

```

[ ]: Best hyperparameters:
{'optimizer': 'adam', 'epochs': 25, 'batch_size': 128}
Test loss: 0.08525428175926208
Test accuracy: 0.9847000241279602
79/79 ██████████ 1s 7ms/step
Best model accuracy: 0.9847
Confusion matrix:
[[ 973    1    0    1    0    0    3    0    1    1]
 [   0 1127    1    2    0    0    1    0    4    0]
 [   3    0 1015    3    1    0    2    4    4    0]
 [   0    0    1  999    0    2    0    2    2    4]
 [   1    0    1    1  965    0    3    1    0  10]
 [   2    0    0    9    0  875    1    1    3    1]
 [   2    2    0    1    2    8  940    0    3    0]
 [   1    2    6    1    1    0    0 1011    3    3]
 [   2    0    1    5    4    2    0    2  953    5]
 [   1    2    0    7    5    2    1    2    0  989]]

```

Bayesian Optimization Algorithm:

```

[ ]: # Load MNIST dataset
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Normalize the grayscale
train_images = train_images / 255.0
test_images = test_images / 255.0

# Convert labels to one-hot encoded format
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

# Define the neural network model

```

```

def create_model(hidden_neurons, learning_rate):
    model = Sequential()
    model.add(Dense(int(hidden_neurons), activation='relu', input_shape=(784,)))
    model.add(Dense(10, activation='softmax'))
    optimizer = SGD(learning_rate=learning_rate)
    model.compile(optimizer=optimizer, loss='categorical_crossentropy',
↳metrics=['accuracy'])
    return model

# Define the objective function for Bayesian Optimization
def objective_function(hidden_neurons, learning_rate):
    model = create_model(hidden_neurons, learning_rate)
    history = model.fit(train_images.reshape(-1, 784), train_labels, epochs=5,
↳batch_size=32, validation_split=0.2, verbose=0)
    validation_accuracy = history.history['val_accuracy'][-1]
    return validation_accuracy

# Initialize the Bayesian Optimization object
optimizer_bayesian = BayesianOptimization(
    f=objective_function,
    pbounds={'hidden_neurons': (10, 50), 'learning_rate': (0.001, 0.01)},
    random_state=42
)

# Perform Bayesian Optimization
optimizer_bayesian.maximize(init_points=10, n_iter=10)

# Get the best hyperparameters
best_hidden_neurons = optimizer_bayesian.max['params']['hidden_neurons']
best_learning_rate = optimizer_bayesian.max['params']['learning_rate']

# Train the model with the best hyperparameters
best_model = create_model(best_hidden_neurons, best_learning_rate)

# Evaluate the best model on the test data
loss, accuracy = best_model.evaluate(test_images.reshape(-1, 784), test_labels,
↳verbose=0)

# Print the test results
print("Test loss:", loss)
print("Test accuracy:", accuracy)

```

Results:

```
[ ]: |  iter  | target | hidden... | learni... |
-----|-----|-----|-----|
|  1    | 0.9388 | 24.98    | 0.009556  |
```

2	0.9304	39.28	0.006388	
3	0.9072	16.24	0.002404	
4	0.9242	12.32	0.008796	
5	0.9348	34.04	0.007373	
6	0.9204	10.82	0.009729	
7	0.9155	43.3	0.002911	
8	0.9078	17.27	0.002651	
9	0.9242	22.17	0.005723	
10	0.9192	27.28	0.003621	
11	0.9285	36.22	0.006602	
12	0.9417	32.07	0.01	
13	0.9076	30.62	0.002191	
14	0.9386	32.84	0.01	
15	0.921	23.98	0.005003	
16	0.8896	25.74	0.001	
17	0.9348	27.67	0.008419	
18	0.9144	10.74	0.004945	
19	0.9298	32.08	0.005957	
20	0.9085	37.61	0.002102	

```
=====
Test loss: 2.4032058715820312
Test accuracy: 0.6524000060558319
```

Genetic Algorithm:

```
[ ]: # Load MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Normalize the grayscale
X_train = X_train / 255.0
X_test = X_test / 255.0

# Define hyperparameter search space
hyperparameters = {
    'hidden_neurons': list(range(10, 51)),
    'learning_rate': [0.001, 0.01, 0.1],
    'batch_size': [4, 8, 12, 16]
}

# Create the model
def create_model(hidden_neurons, learning_rate, batch_size):
    model = Sequential()
    model.add(Dense(hidden_neurons, activation='relu', input_shape=[784]))
    model.add(Dense(10, activation='softmax'))
    optimizer = SGD(learning_rate=learning_rate)
    model.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
```

```

    model.batch_size = batch_size
    return model

# Evaluate model performance
def evaluate_model(model):
    model.fit(X_train.reshape(-1, 784), y_train, epochs=5, batch_size=model.
    ↪batch_size, validation_split=0.2)
    y_pred = np.argmax(model.predict(X_test.reshape(-1, 784)), axis=-1)
    accuracy = accuracy_score(y_test, y_pred)
    return accuracy

# Genetic Algorithm
population_size = 5
num_generations = 2
def initialize_population():
    population = []
    for _ in range(population_size):
        individual = {param: random.choice(values) for param, values in ↪
    ↪hyperparameters.items()}
        population.append(individual)
    return population

def evaluate_individual(individual):
    model = create_model(**individual)
    accuracy = evaluate_model(model)
    return accuracy

def crossover(parent1, parent2):
    child = {}
    for param in hyperparameters.keys():
        child[param] = parent1[param] if random.random() < 0.5 else ↪
    ↪parent2[param]
    return child

def mutate(individual):
    for param in hyperparameters.keys():
        if random.random() < 0.1: # Mutation rate
            individual[param] = random.choice(hyperparameters[param])
    return individual

# Main loop for Genetic Algorithm
population = initialize_population()
best_individual_genetic = None
best_accuracy_genetic = 0.0

for generation in range(num_generations):
    # Evaluate population

```

```

for individual in population:
    accuracy = evaluate_individual(individual)
    if accuracy > best_accuracy_genetic:
        best_accuracy_genetic = accuracy
        best_individual_genetic = individual
print(f"Generation {generation + 1}: Best Accuracy = {best_accuracy_genetic}, Best Individual = {best_individual_genetic}")







# Selection, crossover, and mutation
new_population = []
while len(new_population) < population_size:
    parent1, parent2 = random.sample(population, 2)
    child = crossover(parent1, parent2)
    child = mutate(child)
    new_population.append(child)
population = new_population

# Train and evaluate the best individual found by Genetic Algorithm
print("Best Hyperparameters (Genetic Algorithm):", best_individual_genetic)
best_model_genetic = create_model(**best_individual_genetic)
best_model_genetic.fit(X_train.reshape(-1, 784), y_train, epochs=5,
    batch_size=best_individual_genetic['batch_size'], validation_split=0.2)
y_pred_genetic = np.argmax(best_model_genetic.predict(X_test.reshape(-1, 784)),
    axis=-1)
accuracy_genetic = accuracy_score(y_test, y_pred_genetic)
print("Test Accuracy (Genetic Algorithm):", accuracy_genetic)

```

Results:

```

[ ]: Epoch 1/5
3000/3000  5s 2ms/step - accuracy: 0.8632 - loss: 0.4585 -
    val_accuracy: 0.9461 - val_loss: 0.1790
Epoch 2/5
3000/3000  10s 1ms/step - accuracy: 0.9518 - loss: 0.1600 -
    val_accuracy: 0.9599 - val_loss: 0.1320
Epoch 3/5
3000/3000  4s 1ms/step - accuracy: 0.9651 - loss: 0.1134 -
    val_accuracy: 0.9660 - val_loss: 0.1188
Epoch 4/5
3000/3000  6s 2ms/step - accuracy: 0.9715 - loss: 0.0957 -
    val_accuracy: 0.9685 - val_loss: 0.1111
Epoch 5/5
3000/3000  9s 2ms/step - accuracy: 0.9769 - loss: 0.0756 -
    val_accuracy: 0.9661 - val_loss: 0.1194
313/313  1s 2ms/step
Test Accuracy (Genetic Algorithm): 0.9684

```



Commentary

Random Search:

Test Accuracy: 0.9847

Interpretation: Random Search has performed exceptionally well, achieving a test accuracy of 98.47

Bayesian Optimization:

Test Accuracy: 0.6524

Interpretation: Bayesian Optimization resulted in a significantly lower test accuracy of 65.24

Genetic Algorithm:

Test Accuracy: 0.9684

Interpretation: The Genetic Algorithm achieved a test accuracy of 96.84

Considerations and Conclusion:

Hyperparameters: It's important to consider which hyperparameters were explored and the search space used for each optimization algorithm.

Computational Efficiency: While Random Search can be computationally expensive, it appears to have performed the best in this case.

Robustness: Genetic Algorithm, while not providing the highest accuracy, has still performed well and is more robust in finding good hyperparameter combinations.

Considering the results and the considerations mentioned, it seems that Random Search was the most effective optimization algorithm for this project, followed by the Genetic Algorithm. Bayesian Optimization might not have explored the hyperparameter space as effectively in this scenario. However, considering computational efficiency and robustness, Genetic Algorithm could be preferred over Random Search.