AutoMLP-GA: Automating MLP Optimization with GA

Global and Multi-objective Optimization project

Valeria Insogna
Università di Trieste

January 19 2024

Outline

Project Motivation

Automating MLP design for classification problems using Genetic Algorithms.

Goal

Optimizing MLP topology, type of activation function, and some hyperparameters on MNIST dataset.

Methodology

Discrete GA Framework

Genetic Representation

Individuals in GA population represented by 8 genes that take discrete values.

Parameter	**Values**
Number of Neurons	128, 256, 384, 512, 640, 768, 896, 1024
Number of Layers	1, 2, 3, 4, 5, 6
Activation Function	ReLU, ELU, Tanh, LeakyReLU, Sigmoid
Optimizer	adam, adamw, sgd, rmsprop
Learning Rate Scheduler	cosine, exponential, linear, none
Initial Learning Rate	0.1, 0.01, 0.001, 0.0003, 0.0001
Batch Size	32, 64, 128, 256
Dropout	0, 0.1, 0.2, 0.3, 0.4

Table: Search Space for GA Parameters

Maximization problem

The objective is to maximize **classification accuracy** on the MNIST test set, reflecting the performance of each MLP:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \tag{1}$$

Methodology

GA Operators

Selection

- **Truncated Selection:** 40% of the top networks are retained each generation.
- **Random Selection:** 10% of others are uniformly at random selected.

Crossover

- **Uniform Crossover:** Child networks randomly inherit each parameter from one of the two parents, treating each gene independently.
- $p_{cross} = 100\%$

Mutation

- **Random Mutation:** uniformly at random alternate one of the network's parameters within its range with $p_{mut} = 20\%$

Why Brute-Force is Impractical

The total number of combinations in the search space is calculated as follows. Options for:

- Number of Neurons: 8
- Number of Layers: 6
- Activation Functions: 5
- Optimizers: 4
- Learning Rate Schedulers: 4
- Initial Learning Rates: 5
- Batch Sizes: 4
- Dropout: 5

Total Combinations $= 8 \times 6 \times 5 \times 4 \times 4 \times 5 \times 4 \times 5 = 384,000$

Methodology

Limitations of Brute-Force in MLP Optimization

Advantages of Genetic Algorithm

- Efficiently navigates the vast search space.
- Balances exploration and exploitation.
- Takes less time.

MNIST

- **Dataset Type:** Collection of handwritten digits.
- **Purpose:** Widely used for training and testing in ML.
- **Image Specifications:**
  - Size: 28x28 pixels (784 flattend).
  - Color: Grayscale.
  - Format: Each pixel value is a grayscale intensity between 0 and 255.
- **Training Set:** 60,000 samples.
- **Test Set:** 10,000 samples.
- **Classes:** 10 (Digits 0 to 9).

Github repo AutoMLP-GA:

- **main.py:** Entry point for running the GA-based MLP optimization.
- **network.py:** Defines the Network class, representing the MLPs.
- **optimizer.py:** Implements the Optimizer class for managing the GA.
- **train.py:** Contains functions for training the MLPs using PyTorch.
- **utils.py:** Provides utility functions for result visualization and data management.
- **process.py:** Processes and visualizes results from the baseline algorithm.
- **random_gen.py:** Implements the baseline algorithm by generating and evaluating random networks.

Implementation

`main.py`

Execution Command

`python main.py -gen 10 -pop 20`

where -gen specifies the number of generations, and -pop sets the population size.

```
nn_param_choices = {
    'nb_neurons': [128, 256, 384, 512, 640, 768, 896, 1024],
    'nb_layers': [1, 2, 3, 4, 5, 6],
    'activation': ['ReLU', 'ELU', 'Tanh', 'LeakyReLU', 'Sigmoid'],
    'optimizer': ['adam', 'adamw', 'sgd', 'rmsprop'],
    'lr_scheduler': ['cosine', 'exponential', 'linear', 'none'],
    'initial_lr': [0.1, 0.01, 0.001, 0.0003, 0.0001],
    'batch_size': [32, 64, 128, 256],
    'dropout': [0, 0.1, 0.2, 0.3, 0.4]
}
```

Figure: Visualization of neural network parameters.

Implementation

Network Class

Network Class

Defines the NN architecture in PyTorch, configurable by genetic encoding.

```python
class Network():
    """Represent a network and let us operate on it.

    This is designed for a simple feedforward neural network MLP.
    """

    def __init__(self, nn_param_choices=None):
        """Initialize our network.

        Args:
            nn_param_choices (dict): Parameters for the network, includes:
                nb_neurons (list): [128, 256, 512, 768, 1024]
                nb_layers (list): [1, 2, 3, 4]
                activation (list): ['ReLU', 'ELU', 'Tanh']
                optimizer (list): ['adamw']
        """
        self.accuracy = 0.
        self.nn_param_choices = nn_param_choices
        self.network = {}  # (dict): represents MLP network parameters
        self.model = None

    def create_random(self):
        """Create a random network."""
        for key in self.nn_param_choices:
            self.network[key] = random.choice(self.nn_param_choices[key])
        self.create_network()
```

```python
def create_network(self):
    """Construct a PyTorch network from our parameters."""
    layers = []
    input_size = 784  # MNIST images are 28x28 pixels

    # Creating layers based on the network parameters
    for i in range(self.network['nb_layers']):
        layers.append(nn.Linear(input_size, self.network['nb_neurons']))
        layers.append(getattr(nn, self.network['activation'])())
        layers.append(nn.Dropout(self.network['dropout']))  # Fixed dropout value
        input_size = self.network['nb_neurons']

    # Output layer
    layers.append(nn.Linear(self.network['nb_neurons'], 10))  # 10 classes for MNIST

    self.model = nn.Sequential(*layers)
```

Figure: Left: Code structure of Network class. Right: Schematic of MLP architecture.

Implementation

Optimizer Class

Optimizer Class

- Manages the GA evolutionary process.
- `create_population` initiates the population with diverse network configurations.
- Handles breeding (`breed` method) and mutation (`mutate` method).

```
class Optimizer():
    """Class that implements genetic algorithm for MLP optimization."""

    def __init__(self, nn_param_choices, retain=0.4,
                 random_select=0.1, mutate_chance=0.2):
        """Create an optimizer.

        Args:
            nn_param_choices (dict): Possible network parameters
            retain (float): Percentage of population to retain after
                each generation
            random_select (float): Probability of a rejected network
                remaining in the population
            mutate_chance (float): Probability a network will be
                randomly mutated
        """
        self.mutate_chance = mutate_chance
        self.random_select = random_select
        self.retain = retain
        self.nn_param_choices = nn_param_choices
```

Figure: Optimizer class.

Implementation

Optimizer Class

```python
def evolve(self, pop):
    """Evolve a population of networks.

    Args:
        pop (list): A list of network objects

    Returns:
        (list): The evolved population of networks
    """
    graded = [(self.fitness(network), network) for network in pop]
    graded = [x[1] for x in sorted(graded, key=lambda x: x[0], reverse=True)]

    retain_length = int(len(graded) * self.retain)
    parents = graded[:retain_length]

    for individual in graded[retain_length:]:
        if self.random_select > random.random():
            parents.append(individual)

    parents_length = len(parents)
    desired_length = len(pop) - parents_length
    children = []
  Valeria, 2 weeks ago • Create optimizer
    while len(children) < desired_length:
        male = random.randint(0, parents_length-1)
        female = random.randint(0, parents_length-1)

        if male != female:
            male = parents[male]
            female = parents[female]
            babies = self.breed(male, female)

            for baby in babies:
                if len(children) < desired_length:
                    children.append(baby)

    parents.extend(children)
    return parents
```

```python
def mutate(self, network):
    """Randomly mutate one part of the network.

    Args:
        network (Network): The network to mutate

    Returns:
        (Network): A mutated network object
    """
    #print("Mutating network")
    mutation = random.choice(list(self.nn_param_choices.keys()))
    network.network[mutation] = random.choice(self.nn_param_choices[mutation])
    network.create_network()  # Rebuild the network with new mutation
    return network
```

```python
def breed(self, mother, father):
    """Make two children from parts of their parents.

    Args:
        mother (Network): Network object
        father (Network): Network object

    Returns:
        (list): Two network objects
    """
    #print("Breeding networks")
    children = []
    for _ in range(2):

        child = {}

        for param in self.nn_param_choices:
            child[param] = random.choice(
                [mother.network[param], father.network[param]]
            )

        network = Network(self.nn_param_choices)
        network.create_set(child)

        if self.mutate_chance > random.random():
            network = self.mutate(network)

        children.append(network)

    return children
```

Figure: From left to right: selection, mutation and crossover

Training

- Training and evaluation are conducted using the MNIST dataset.
- Performance is measured by accuracy during training iterations.

```python
# Select Optimizer
if network.network['optimizer'] == 'adam':
    optimizer = optim.Adam(model.parameters(), lr=network.network['initial_lr'])
elif network.network['optimizer'] == 'adamw':
    optimizer = optim.AdamW(model.parameters(), lr=network.network['initial_lr'])
elif network.network['optimizer'] == 'sgd':
    optimizer = optim.SGD(model.parameters(), lr=network.network['initial_lr'], momentum=0.9)
elif network.network['optimizer'] == 'rmsprop':
    optimizer = optim.RMSprop(model.parameters(), lr=network.network['initial_lr'])

# Select Learning Rate Scheduler
if network.network['lr_scheduler'] == 'cosine':
    scheduler = CosineAnnealingLR(optimizer, T_max=100)
elif network.network['lr_scheduler'] == 'exponential':
    scheduler = ExponentialLR(optimizer, gamma=0.9)
elif network.network['lr_scheduler'] == 'linear':
    scheduler = LinearLR(optimizer)
else:  # 'none'
    scheduler = None
```

```python
# Training Loop
for epoch in range(10):  # Loop over the dataset multiple times
    if debug:
        print(f"Epoch {epoch+1}/10")
    for _, data in enumerate(train_loader, 0):
        if debug:
            print(f"Batch {_+1}/{len(train_loader)}")
        inputs, labels = data
        inputs = inputs.view(inputs.size(0), -1)  # Flatten the images
        optimizer.zero_grad()

        outputs = model(inputs)
        loss = criterion(outputs, labels) # CrossEntropyLoss
        loss.backward()
        optimizer.step()

        if scheduler:
            scheduler.step()
```

Figure: Left: Training parameters visualization. Right: Training process visualization.

Results

First Run

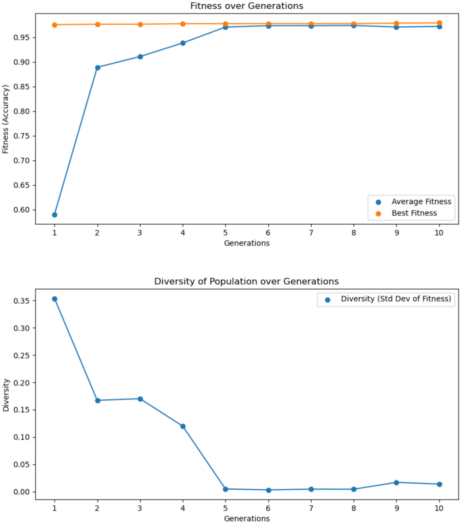
Evolution Summary

Over 10 generations with a population of 50. GA showed consistent improvement in network accuracy over successive generations.

Generation	Average Accuracy
1	59.03%
2	88.92%
3	91.09%
4	93.87%
5	97.09%
6	97.36%
7	97.37%
8	97.42%
9	97.10%
10	97.23%

Results

First Run



Results

First Run

Top Networks

Configurations of the top 10% performing networks along with their achieved accuracies.

#	Neurons	Layers	Activ	Optimizer	Sched	Accuracy
1	1024	1	ReLU	adamw	cosine	97.94%
2	1024	3	ReLU	rmsprop	cosine	97.90%
3	1024	3	Sigmoid	adam	none	97.79%
4	1024	1	Sigmoid	rmsprop	cosine	97.78%
5	1024	1	Sigmoid	rmsprop	cosine	97.77%

Table: Configuration details of the top 5 networks.

Computation Time

On MacBook M1 it took on average 40s for each network to train.

```
(DL) valeriainsogna@MacBook-Air-di-Valeria AutoMLP-GA % python main.py --gen 7 --pop 50
100%|                                        | 50/50 [55:28<00:00, 66.58s/it]
100%|                                        | 50/50 [21:26<00:00, 25.73s/it]
100%|                                        | 50/50 [43:28<00:00, 52.17s/it]
100%|                                        | 50/50 [18:51<00:00, 22.63s/it]
100%|                                        | 50/50 [22:30<00:00, 27.01s/it]
100%|                                        | 50/50 [16:06<00:00, 19.33s/it]
100%|                                        | 50/50 [22:26<00:00, 26.92s/it]
```

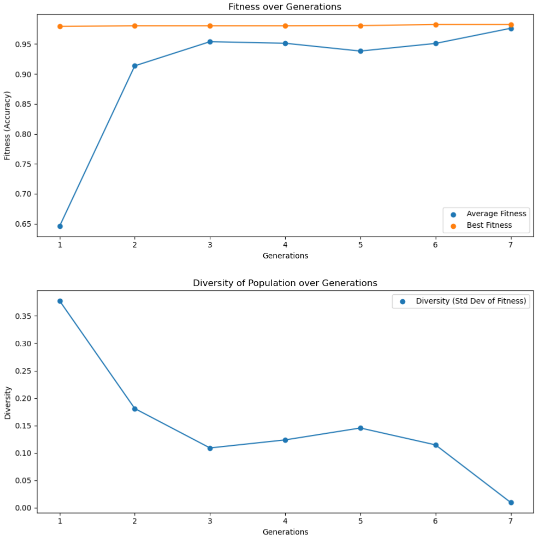
Evolution Summary

Over 7 generations with a population of 100, the GA demonstrated significant improvement in network accuracy, with the final generation achieving an average accuracy of 97.63

Generation	Average Accuracy
1	64.58%
2	91.36%
3	95.39%
4	95.13%
5	93.82%
6	95.10%
7	97.63%

Table: Generation-wise average accuracy.

Results
Second Run



Top Performing Networks

The top 10% networks from the final generation showcased impressive accuracies, with the best network achieving 98.26%.

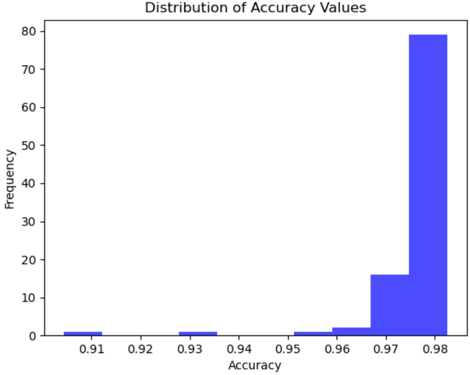
Neurons	Layers	Activation	Optimizer	Scheduler	Initial LR	Batch Size	Dropout	Accuracy
1024	3	LeakyReLU	adamw	linear	0.0001	64	0	98.26%
1024	3	LeakyReLU	adam	linear	0.0001	32	0	98.23%
768	3	LeakyReLU	adamw	none	0.0001	32	0	98.22%
768	3	LeakyReLU	adamw	none	0.0001	32	0	98.20%
768	2	LeakyReLU	adam	cosine	0.0003	32	0	98.14%
768	3	LeakyReLU	adamw	cosine	0.0001	32	0	98.09%
896	3	LeakyReLU	adamw	cosine	0.0001	32	0	98.07%
768	3	ReLU	adamw	linear	0.0003	32	0	98.06%
896	1	LeakyReLU	sgd	linear	0.01	32	0	98.06%
768	3	LeakyReLU	adamw	linear	0.0001	32	0	98.06%

Table: Specifications of the top-performing networks.

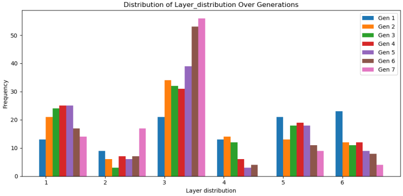
Results

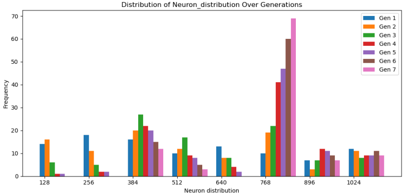
Second Run

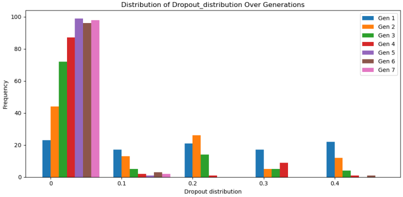
Fitness distribution in final population:
Mean: 0.9763, Median: 0.9791, Std: 0.0096

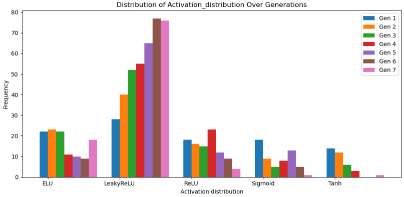


Results
Second Run

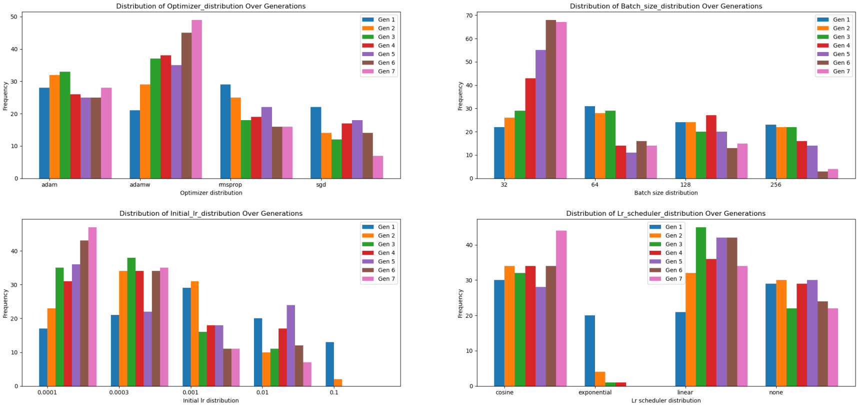








Results
Second Run



Computation Time

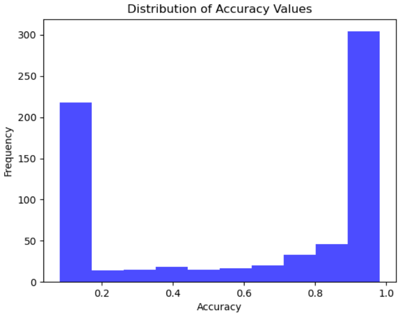
On MacBook M1 it took on average 40s for each network to train.

```
(DL) valeriainsogna@MacBook-Air-di-Valeria AutoMLP-GA % python main.py --gen 7 --pop 100
100%|██████████| 100/100 [1:37:39<00:00, 58.59s/it]
100%|██████████| 100/100 [47:50<00:00, 28.71s/it]
100%|██████████| 100/100 [52:44<00:00, 31.64s/it]
100%|██████████| 100/100 [1:04:33<00:00, 38.73s/it]
100%|██████████| 100/100 [1:02:09<00:00, 37.29s/it]
100%|██████████| 100/100 [1:24:40<00:00, 50.80s/it]
100%|██████████| 100/100 [1:08:31<00:00, 41.11s/it]
(DL) valeriainsogna@MacBook-Air-di-Valeria AutoMLP-GA %
```

Baseline Comparison

Compare results with random creation of same number of $gen * pop = 700$ networks.

Fitness distribution:
Mean: 0.6043, Median: 0.8015, Std: 0.3775



Top 10 Networks by Accuracy

Accuracy	Neurons	Layers	Activ	Optimizer	LR Sched	Initial LR	Batch	Dropout
98.25%	640	2	LeakyReLU	adam	none	0.0003	32	0
98.03%	1024	3	LeakyReLU	rmsprop	none	0.0001	32	0.1
98.02%	384	3	LeakyReLU	adam	none	0.0003	32	0
98.02%	512	3	LeakyReLU	rmsprop	cosine	0.0003	32	0
98.02%	768	1	Tanh	adam	none	0.0003	32	0
97.93%	768	1	LeakyReLU	adam	cosine	0.001	128	0
97.93%	1024	4	ELU	sgd	none	0.01	32	0
97.92%	1024	2	ReLU	adamw	linear	0.001	256	0.1
97.91%	1024	3	ReLU	rmsprop	linear	0.0001	64	0.1
97.90%	256	6	LeakyReLU	rmsprop	cosine	0.0003	32	0

Table: Top 10 Networks sorted by Accuracy

Innovative Approach

- Configure the network by selecting a random number of layers within the maximum limit defined in `nb_max_layers`, a new gene.
- It assigns a number of neurons for each of these layers.
- Parameters `neurons_per_layer_i` that are not used (for inactive layers) are set to `None`.

```
"""
nn_param_choices (dict): Parameters for the network, includes:
    nb_max_layers (list): [1, 2, 3, ..., n]  # Example values
    neurons_per_layer_1 (list): [64, 128, 256, ...]
    neurons_per_layer_2 (list): [64, 128, 256, ...]
    ...
    neurons_per_layer_n (list): [64, 128, 256, ...]
```

Figure: Flexible configuration of neural network layers.

Dynamic Layer Configuration

The `create_random` function will now configure the network with a variable number of layers and assign neurons to each layer within the maximum limit defined by `nb_max_layers`.

```python
def create_random(self):
    """Create a random network."""
    # Seleziona un numero casuale di strati
    num_layers = random.choice(self.nn_param_choices['nb_max_layers'])
    for i in range(1, max(self.nn_param_choices['nb_max_layers']) + 1):
        if i <= num_layers:
            self.network[f'neurons_per_layer_{i}'] = random.choice(self.nn_param_choices[f'neurons_per_layer_{i}'])
        else:
            self.network[f'neurons_per_layer_{i}'] = None

    # Seleziona altri parametri della rete
    for param in self.nn_param_choices:
        if 'neurons_per_layer' not in param and param != 'nb_max_layers':
            self.network[param] = random.choice(self.nn_param_choices[param])

    self.create_network()
```

Figure: Dynamic selection of layers and neurons in the `create_random` method.

Network Layer Variability

Each network can have a random number of layers with the number of neurons per layer also being randomly assigned, leading to a more diverse set of network architectures, potentially exploring more complex solutions.

```python
def create_network(self):
    """Construct a PyTorch network from our parameters."""
    layers = []
    input_size = 784  # Ad esempio, per immagini MNIST che sono 28x28 pixels

    for i in range(1, max(self.nn_param_choices['nb_max_layers']) + 1):
        # Aggiungi solo gli strati che sono stati selezionati in create_random
        if self.network[f'neurons_per_layer_{i}']:
            # Aggiungi un strato lineare
            layers.append(nn.Linear(input_size, self.network[f'neurons_per_layer_{i}']))
            # Aggiungi un'attivazione (se specificata)
            if 'activation' in self.network:
                layers.append(get_activation(self.network['activation'])())
            # Aggiungi il dropout (se specificato)
            if 'dropout' in self.network:
                layers.append(nn.Dropout(self.network['dropout']))
            # Aggiorna la dimensione dell'input per il prossimo strato
            input_size = self.network[f'neurons_per_layer_{i}']

    # Aggiungi lo strato di output
    # Assumiamo che il numero di classi di output (ad es. per MNIST) sia noto
    output_classes = 10  # Ad esempio, 10 classi per MNIST
    layers.append(nn.Linear(input_size, output_classes))

    # Costruisci il modello finale
    self.model = nn.Sequential(*layers)
```

Figure: Construction of a PyTorch network with flexible layer and neuron counts.

Adaptive Breeding Process

The breeding process will be enhanced to handle the dynamic structures, ensuring that children networks inherit the adaptable traits of their parents.

```
def breed(self, mother, father):
    """Make a child from parts of both parents."""
    child = {}
    # Scegli casualmente i parametri da uno dei genitori
    for param in self.nn_param_choices:
        child[param] = random.choice([mother.network[param], father.network[param]])

    # Fix per nb_max_layers e neurons_per_layer_i
    child_nb_layers = child['nb_max_layers']
    for i in range(1, max(self.nn_param_choices['nb_max_layers']) + 1):
        param_name = f'neurons_per_layer_{i}'
        if i > child_nb_layers:
            # Imposta a None per strati non utilizzati
            child[param_name] = None
        elif child[param_name] is None:
            # Se neurons_per_layer_i è None ma dovrebbe essere usato, scegli un valore di default
            child[param_name] = random.choice(self.nn_param_choices[param_name])

    return Network(self.nn_param_choices).create_set(child)
```

Figure: Adaptive breeding function accommodating flexible network structures.

Conclusion
Future Improvements

- `epoch` as a discrete gene
- add dropout only between some layers
- univariate EDA (PBIL) with mix of discrete/real values for genes.