Computational Intelligence ( )

Submission Assignment 5

Instructor: JakubM. Tomczak *Name:* [Axel Ehrnrooth], *Netid:* [aeh330]

1 Introduction

An artificial neural network (NN) is a powerful tool for handling complex machine-learning problems such as image recognition and classification. The performance of a NN depends heavily on its complexity and overall architecture, both of which can vary immensely depending on the task at hand. Even when tackling a single problem, good design choices are a quintessential part of ensuring optimal performance. However, because of the virtually unlimited variations available for neural network design, the model space becomes very large and optimising the architecture by brute force is inefficient even for a small network. This is where Neuroevolutionary algorithms can become extremely useful.

A Neural Architecture Search (NAS) algorithm can automate the process of optimising the design of an artificial neural network. The NAS algorithm, as its name suggests, uses features derived from natural evolution, where individuals evolve and mutate over generations, throughout which only the best performers survive. Applying these principles to the architecture of a NN allows for fast and accurate optimisation in an enormous search space of models.

The NAS discussed in this report is specifically designed to optimise a convolutional neural network (CNN) for image recognition. The overall structure of the network (e.g. the number of convolutional and linear layers) is determined ahead of time, and the hyperparameters that the network uses are optimised through the NAS algorithm, which runs for a specified number of generations. In each generation, all current network configurations are trained on the same data set and evaluated to select the best candidates for survival.

2 Problem statement

The purpose of designing a neural architecture search algorithm is to automate the discovery of optimal neural network architectures. This process usually involves optimising the structure and depth of the network or exploring the performance of different hyperparameters. In general, the goal of the NAS is to minimise a specific function associated with a neural network (usually an objective function such as the loss), but the true goal may vary depending on the task. More often than not, the aim is to optimise the accuracy of the neural network.

The decision to implement a NAS algorithm for optimisation stems from the need to search through a large model space. Manually optimising the architecture of a neural network becomes time-consuming due to the exponentially growing model space with respect to the number of modifiable parameters. Automating the process of architecture optimisation leads to increased efficiency and high-performance models.

3 Methodology

3.1 Convolutional Neural Network (CNN)

This section will explain all components that make up the CNN used in this project. As mentioned before, the general structure of the network is predefined as follows:

1. 2-Dimensional convolutional layer
2. Activation function
3. Pooling layer
4. Linear layer
5. Activation function

6. Output layer
7. Log-Softmax activation

The hyperparameters – such as the activation functions and the number of filters – that are passed into the CNN are initially determined using a randomly generated seed. The seed is decoded using a dictionary which contains lists of all possible hyperparameter options for each component. The convolutional layer varies in terms of the number of filters it produces (8, 16, or 32) as well as the kernel size (3x3 and 5x5) and padding amount (1 or 2). It is important to note, however, that the smaller kernel size is only matched with the smaller padding and vice versa; the kernel size cannot be 3 when the padding is 2. The activation function can either be a ReLU, Sigmoid, Tanh, or ELU function, all of which can be seen below.

$$\text{ReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

$$\text{ELU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(e^{x} - 1) & \text{if } x < 0 \end{cases}$$

The pooling has two parameters, one of which is the kind of pooling (average or max pooling), and the other is the size (2x2 or identity). The linear layer varies in the number of neurons it contains, which can either be 10, 20, 30, 40, 50, 60, 70, 80, 90, or 100.

The objective function used for the network is the negative log-likelihood (NLL) loss function, which can be seen below.

$$loss(x, y) = \sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{y_n}} l_n$$

(1)

The reason that NLL was used instead of cross-entropy loss (which is a more common objective function for multi-class classification algorithms) is because the performance of the final network will be compared to two other networks that use NLL loss which were trained and tested on the same data as the networks in this case.

For the output of the neural network, the log-softmax function is used, which can be seen below.

$$\text{logsoftmax}(x_i) = \log\left(\frac{e^{x_i}}{\sum_j e^{x_j}}\right)$$

Even though using just a softmax function will yield the exact same classification results, the log-softmax operates in a logarithmic space which yields more numerically stable results. Either way, the probabilities of each class are proportional to the probabilities when using the regular softmax function.

Ideally, a network should be trained on the same data set over many epochs. Nevertheless, increasing the number of epochs causes computational requirements to rise. Since the evolutionary algorithm will be training multiple networks with various configurations, the number of epochs is drastically lowered to ten.

3.2 Neural Architecture Search (NAS)

The evolutionary algorithm used for this task consists of five components: parent selection, recombination, mutation, evaluation, and survivor selection. The algorithm runs for a total of 50 generations.

The parent selection works by selecting ten individuals from the population in a random order and returning this list of individuals as parents. The list of parents is then used for recombination, where a simple one-point crossover between two parents is performed.

The crossover point is selected randomly for each pair of parents. Each pair of parents will produce two children, one of which will have the attributes of its father that occur before the cross-over point and that of its mother that occur at and after the cross-over point. The opposite will be true for the second child. An illustration of this can be seen below.

Cross-over point = 3
Parents: [f,f,f,f,f] – [m,m,m,m,m]
Children: [f,f,m,m,m] – [m,m,f,f,f]

The children are then added to the population, resulting in a total of 20 individuals.

Once the children have been added, half of the population will undergo a mutation of a single feature. The feature that is up for mutation is chosen randomly. The mutation will help introduce features that may not have existed in the original population or re-introduce features that were lost in some generations.

Once the mutations have occurred, the entire population is evaluated. The evaluation entails training 20 neural networks over ten epochs, each with all different configurations existing in the population. Each network will have a fitness value based on the classification error of the network on the testing data. The objective of the evolutionary algorithm is to minimise the fitness value, which is calculated using the following formula:

$$\text{Objective} = \text{ClassError} + \lambda \frac{N_P}{N_{max}}$$

Where $\lambda = 0.01$ and N_P is the number of weights in the network.

This function adds a penalty for larger networks. The reason for this is that we want to encourage smaller networks over larger ones as the computational requirements increase with the size of the networks. The lower the fitness value of the network is, the higher its chances of survival will be.

Once the entire population has been evaluated, the individuals are ordered in terms of fitness value from lowest to highest. The ten best-performing individuals are then selected as the surviving generation and will be selected for further recombination.

It is important to note that the seed mentioned earlier in the report is only used to initialise the population. Once the initial ten individuals exist, the evolutionary algorithm takes over and optimises the hyperparameters based on the already existing population.

4 Experiments

The purpose of the experimentation is to show a proof of concept that a NAS algorithm can optimise a CNN to outperform manually optimised deeper neural networks. To do this, the networks are trained on the *Digits* data set from scikit-learn.(2) This is a simple data set of 8x8 images with a single dimension representing the grey scale value of each pixel. The simplicity and size of the data set allow the networks to train relatively quickly, which can be useful for gaining insight into the NAS's capabilities before running it on more complex data.

The NAS runs for 50 generations. In each generation, it randomly orders the population of parents for recombination and doubles the population size when producing children. The children are based on 1-point crossover recombination. The entire population is subject to the mutation, but only half the candidates will end up with a mutation. While mutation can introduce better features, withholding 50% of the candidates halves the risk of removing existing high-performance features.

Because the NAS trains 20 networks per generation, longer training times can exponentially increase the running time of the algorithm. To tackle this issue, networks train only for ten epochs (if there is no improvement after three consecutive epochs, the network stops training). While reducing the training time will help with overall run times, having a low learning rate for high-precision results can be problematic, as ten epochs may not be enough time for a network to finish learning at such a slow rate. For this reason, the learning rate is set to 0.01 instead of the slower and more precise 0.001.

The effectiveness of a specific network configuration is based on the total classification error on the testing data. This gives a reliable measure of the network's ability to classify images and can therefore be analysed and compared to a manually optimised network. Because it is desirable to have a smaller network, the fitness of a network is based on the total number of weights in the network in addition to the classification error in the testing data.

5 Results and discussion

The results of the evolutionary algorithm were promising. As can be seen in Figure 1, the fitness level starts off around 0.07 and continuously improves for the next 18 or so generations until it converges and plateaus around 0.036. The network that yielded these results had the following configuration: **[number of filters: 32;**

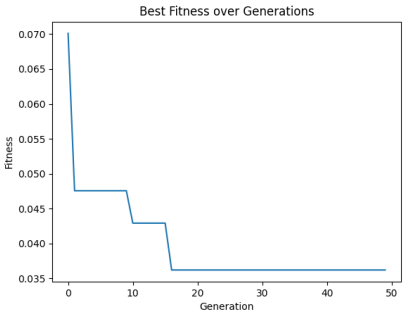


Figure 1: Fitness values over generations (NAS)

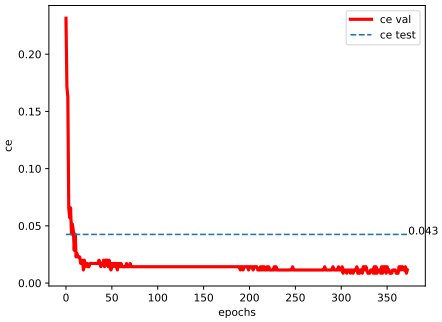


Figure 2: Classification error of standalone CNN

kernel size: 3x3; padding: 1; activation: Tanh; pooling: max-pooling; pool-size: 2x2; number of neurons: 50].

The results of the standalone CNN and MLP can be seen in Figures 2 and 3, respectively. The blue dashed line in the plots represents the classification error on the testing data for those networks. Though it would make sense for networks that are deeper and contain more nodes per layer to perform at a higher level, it can be seen that the optimisation that occurred thanks to the evolutionary algorithm yielded better results. It is important to note, however, that for simpler data sets, more complexity in models can cause problems such as overfitting, which can be detrimental to the final performance of a network. Especially for the MLP, where the error on the testing data is significantly higher than that on the training data, suggests some degree of overfitting.

The goal of the evolutionary algorithm was to find a configuration for a neural network that could outperform other networks that were optimised manually. The true classification error on the testing data of the best-performing network can be derived using the objective function for the evolutionary algorithm:

$$\text{Objective} = \text{ClassError} + \lambda \frac{N_P}{N_{max}}$$

Using this function, the fitness penalty for this configuration ends up being 0.005, meaning that the true classification error on the testing data was 0.031. This error is significantly lower than that of the standalone CNN and is dwarfed in comparison to the classification error of the MLP.

The second goal of the NAS was to search the enormous model space for a good configuration of a CNN. With the configuration options available to the network, the entire model space consists of 4500 unique model configurations. The fact that the NAS runs for 50 generations and trains 20 networks in each generation means that it explored a total of 1000 network configurations (not all configurations are necessarily unique, especially when the fitness value starts to plateau). The fact that the model search shown in Figure 1 reached its optimum value on the 18^{th} generation means that it only had to explore a maximum of 360 configurations before finding a suitable network. Considering that the best configuration was found after only searching 8% of the total model space shows that this goal was also achieved.

Nevertheless, there are some aspects of this NAS model that could be further improved and some aspects added. Firstly, the depth of the network never changes during the optimization. Adding multiple convolutional layers can create better filters for classification. Adding a dropout layer in the fully connected network could also help with removing unwanted biases in the final result.(3) Furthermore, the data that the networks are trained on allows the entire algorithm to run relatively quickly but on larger data sets of bigger images with

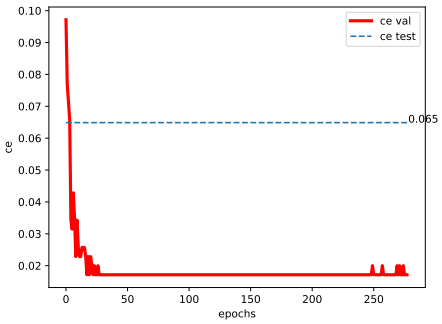


Figure 3: Classification error of MLP

higher dimensions, this implementation can take hours or even days to run on a laptop. Because of this, the networks have to train quickly (over a small number of epochs) not to further increase the running time. Another potential improvement to the evolutionary component of this model could be basing the pairing of parents based on performance. Doing so would ideally result in faster production of top-performing candidates.

References

"torch.nn.NLLLoss - pytorch documentation," https://pytorch.org/docs/stable/generated/torch.nn.NLLLoss.html#torch.nn.NLLLoss, accessed on [22 Apr. 2023].
"load_digits," https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_digits.html, [Online; accessed 22 Apr. 2023].
T. DeVries and G. W. Taylor, "Improved regularization of convolutional neural networks with cutout," *arXiv preprint arXiv:1708.04552*, 2017.

In [ ]:
```
import os

import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset, Subset
from torchvision import transforms, datasets
import random
# from numba import jit, cuda

EPS = 1.0e-7
```

In [ ]:
```
# Digits dataset for testing
# this dataset allows the algorithm to finish faster

from sklearn import datasets
from sklearn.datasets import load_digits

class Digits(Dataset):
    """Scikit-Learn Digits dataset."""

    def __init__(self, mode="train", transforms=None):
        digits = load_digits()
        if mode == "train":
            self.data = digits.data[:1000].astype(np.float32)
            self.targets = digits.target[:1000]
        elif mode == "val":
            self.data = digits.data[1000:1350].astype(np.float32)
            self.targets = digits.target[1000:1350]
        else:
            self.data = digits.data[1350:].astype(np.float32)
            self.targets = digits.target[1350:]

        self.transforms = transforms

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        sample_x = self.data[idx]
        sample_y = self.targets[idx]
        if self.transforms:
            sample_x = self.transforms(sample_x)
        return (sample_x, sample_y)

train_data = Digits(mode="train")
val_data = Digits(mode="val")
test_data = Digits(mode="test")

# Initialize data loaders.
train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
val_loader = DataLoader(val_data, batch_size=64, shuffle=False)
test_loader = DataLoader(test_data, batch_size=64, shuffle=False)
```

In [ ]:
```
# folder to save results in
results_dir = "./results"
```

In [ ]:
```
# # ALL DATA STUFF

# transform = transforms.Compose([
```

```
#     transforms.ToTensor(),
# ])

# # import MNIST dataset
# training_dataset = datasets.MNIST(
#     "data",
#     train=True,
#     download=True,
#     transform=transform
# )

# test_dataset = datasets.MNIST(
#     "data",
#     train=False,
#     download=True,
#     transform=transform
# )

# indices = [i for i in range(len(training_dataset))]
# train_split = int(len(training_dataset) * 0.8)
# train_dataset = Subset(training_dataset, indices[:train_split])
# val_dataset = Subset(training_dataset, indices[train_split:])


# train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
# val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False)
# test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

In [ ]:

```
class Flatten(nn.Module):
    def __init__(self):
        super(Flatten, self).__init__()

    def forward(self, x):
        return x.view(x.shape[0], -1)


class Reshape(nn.Module):
    def __init__(self, size):
        super(Reshape, self).__init__()
        self.size = size  # a list

    def forward(self, x):
        assert x.shape[1] == np.prod(self.size)
        return x.view(x.shape[0], *self.size)
```

In [ ]:

```
class CNN(nn.Module):
    def __init__(self, classnet):
        super(CNN, self).__init__()
        self.classnet = classnet
        self.nll = nn.NLLLoss(reduction='none')

    def classify(self, x):

        y_pred = torch.argmax(self.classnet(x), dim=1)

        return y_pred

    def forward(self, x, y, reduction="avg"):
        # x = x.type(torch.float32)   # Needed to turn the data into float32
                                      # in order to avoid a RuntimeError.
        loss = self.nll(self.classnet(x), y.long())

        if reduction == "avg":
```

```
            return loss.mean()
        elif reduction == "sum":
            return loss.sum()
```

In [ ]:
```
def evaluation(test_loader, name=None, model_best=None, epoch=None):
    if model_best is None:
        model_best = torch.load(name + ".model")

    model_best.eval()
    loss_test = 0.0
    loss_error = 0.0
    N = 0.0

    for indx_batch, (test_batch, test_labels) in enumerate(test_loader):
        loss_test_batch = model_best.forward(test_batch, test_labels, reducti
        loss_test += loss_test_batch.item()

        y_pred = model_best.classify(test_batch)
        e = 1.0 * (y_pred == test_labels)
        loss_error += (1.0 - e).sum().item()

        N += test_batch.shape[0]

    loss_test /= N
    loss_error /= N

    # if epoch is None:
    #     print(f"FINAL PERFORMANCE: nll={loss_test}, ce={loss_error}")
    # else:
    #     if epoch % 2 == 0:
    #         print(f"EPOCH {epoch}: nll={loss_test}, ce={loss_error}")

    return loss_test, loss_error
```

In [ ]:
```
#TRAINING

def training(
        name, max_patience, num_epochs, model, optimizer, train_loader, val_l
):

    nll_val = []
    error_val = []
    best_nll = 1000.0
    patience = 0

    for e in range(num_epochs):
        model.train()

        for indx_batch, (batch, targets) in enumerate(train_loader):

            loss = model.forward(batch, targets)

            optimizer.zero_grad()

            loss.backward(retain_graph=True)

            optimizer.step()

        loss_e, error_e = evaluation(val_loader, model_best=model, epoch=e)
        nll_val.append(loss_e)
        error_val.append(error_e)

        if e == 0:
```

```
            torch.save(model, name + ".model")
            best_nll = loss_e
        else:
            if loss_e < best_nll:
                torch.save(model, name + ".model")
                best_nll = loss_e
                patience = 0
            else:
                patience += 1

        if patience > max_patience:
            break

    nll_val = np.asarray(nll_val)
    error_val = np.asarray(error_val)

    return nll_val, error_val
```

In [ ]:

```
# INITIALIZE THE POPULATION

# This dictionary holds all possible parameters for any CNN in the model spac
dict = {1: [8,16,32],                                   # Number of filters
        2: [(3,1), (5,2)],                              # Kernel size and pad
        3: [nn.ReLU, nn.Sigmoid, nn.Tanh, nn.ELU],      # Activation fucntion
        4: [2, 1],                                      # Pooling
        5: [nn.AvgPool2d, nn.MaxPool2d],                # Pooling
        6: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]    # Number of neurons
        }

def generate_individual(seed):
    """
    Generates a random individual for the evolutionary algorithm.
    """
    # Number of filters
    num_filters = dict[1][seed[0]]
    # Kernel size and padding
    kern_pad = dict[2][seed[1]]
    # Activation
    activation = dict[3][seed[2]]
    # Pooling
    pooling1 = dict[4][seed[3]]
    pooling2 = dict[5][seed[4]]
    # Linear layer
    num_neurons = dict[6][seed[5]]

    return [num_filters, kern_pad, activation, pooling1, pooling2, num_neuron


def generate_seed():
    """
    Generates a random seed for the evolutionary algorithm.
    """
    seed = [] # initialize the seed
    for i in range(6): # generate a random number for each parameter
        seed.append(np.random.randint(0, len(dict[i+1])))
    return seed

seeds = [] # Initialize the list of all seeds
while len(seeds) < 10: # Generate 10 seeds
    seed = generate_seed()
    if seed not in seeds: # Make sure the seed does not already exist
        seeds.append(seed) # Add the seed to the list of seeds

population = [] # Initialize the population
```

```
for seed in seeds:
    population.append(generate_individual(seed)) # Generate an individual for
```

In [ ]:

```
result_dir = "./results"

# Training hyperparameters
lr = 1e-2 # learning rate
wd = 1e-5 # weight decay
num_epochs = 10  # number of epochs (max)
max_patience = 3 # patience for early stopping


class Network:
    def __init__(self, population):
        """
        Import the population and initialize all constants.
        """
        self.population = population
        self.H = 8                      # Height of the image
        self.W = 8                      # Width of the image   --> (28x28 - MN
        self.D = 1                      # Input dimension     --> (1 - MNIST,
        self.K = 10                     # Number of classes

    def run(self):
        """
        Run the network on each individual in the population.
        """
        iter = -1 # Initialize the iteration number
        fitness = [] # Initialize the list of fitness values
        for i in range(len(self.population)):
            iter += 1  # Increment the iteration number
            name = "classifier_cnn" # Initialize the name of the model
            name += "_"
            name += str(iter)  # Specify which cnn we are on
            # print(name)

            # Initialize the model
            num_kernels = self.population[i][0] # Number of kernels
            kernel_size = self.population[i][1][0] # Kernel size
            padding = self.population[i][1][1] # Padding
            activation = self.population[i][2] # Activation function
            pooling1 = self.population[i][3] # Pooling
            pooling2 = self.population[i][4] # Pooling
            num_neurons = self.population[i][5] # M
            l_in = ((self.W * self.H * num_kernels) // pooling1)// pooling1 #
            self.num_weights = (num_kernels * kernel_size * kernel_size * sel
            self.weights_max = 628320 # maximum weight value


            size = [1,8,8] # 8's are ignored when not using Digits
            classnet = nn.Sequential(

                # Reshape the input to the correct shape for the convolutiona
                Reshape(size),

                # Convolutional layer
                nn.Conv2d(size[0], num_kernels, kernel_size=kernel_size, stri
                activation(), # Activation function

                # Pooling layer
                pooling2(pooling1, pooling1),

                # Flatten the output for the linear layer
                Flatten(),
```

```
                # Linear layer
                nn.Linear(l_in, num_neurons),
                activation(),

                # Output layer
                nn.Linear(num_neurons, self.K),
                nn.LogSoftmax(dim=1),

            )

            model = CNN(classnet) # define the model

            optimizer = torch.optim.Adamax(
                [p for p in model.parameters() if p.requires_grad == True],
                lr=lr,
                weight_decay=wd
            )

            # train the model
            nll_val, error_val = training(
                name=result_dir + name,
                max_patience=max_patience,
                num_epochs=num_epochs,
                model=model,
                optimizer=optimizer,
                train_loader=train_loader,
                val_loader=val_loader,
            )

            # run test
            test_loss, test_error = evaluation(name=result_dir + name, test_l
            f = open(result_dir + name + "_test_loss.txt", "w")
            f.write("NLL: " + str(test_loss) + "\nCE: " + str(test_error))
            f.close()


            # calcluate fitness value
            # fitness is calculated using the test error and the number of ne
            fitness.append((test_error+(lr*((self.num_weights)/self.weights_m
            # equation: fitness = test_error + (lr * (Np/Nmax))
            # Np = number of neurons in the linear layer
            # Nmax = maximum number of neurons in the linear layer
            # lr = learning rate (0.01)

        # print the two lowest fitness values
        #fitness.sort(key=lambda x: x[0])
        #print ("Fitness: ", fitness[0:2])
        return fitness
```

In [ ]:

```
class EvolutionaryAlgorithm:
    def __init__(self, network, population):
        self.network = network # initialize the network
        self.pop_size = len(population) # population size
        self.n_parents = self.pop_size # number of parents
        # bounds for mutation
        self.bounds_max = [len(dict[1]),
                           len(dict[2]),
                           len(dict[3]),
                           len(dict[4]),
                           len(dict[5]),
                           len(dict[6])] # upper bound for each parameter
```

```
        self.bounds_min = [0, 0, 0, 0, 0, 0] # lower bound for each parameter

    def parent_selection(self, x_old):
        # select n parents randomly from the population
        x_parents = random.sample(list(x_old), self.n_parents)

        return x_parents

    def recombination(self, x_parents):
        """
        Random Cross-over
        """
        x_children = [] # initialize the children

        # select random crossover point
        cross_over = np.random.randint(0, len(x_parents[0]))


        for i in range(self.n_parents//2):
            # select two random parents
            parents = random.sample(list(x_parents), 2)
            parent1 = parents[0] # define parent 1
            parent2 = parents[1] # define parent 2

            # combine parent 1 and parent 2 at crossover point
            child1 = np.concatenate((parent1[:cross_over], parent2[cross_over
            child2 = np.concatenate((parent2[:cross_over], parent1[cross_over

            #add child1 and child2 to children
            x_children.append(child1)
            x_children.append(child2)

        # add children to population
        x_children = np.concatenate([x_children, x_parents])

        return x_children

    def mutation(self, x_children):
        """
        Random mutation
        """
        for i in range(int(len(x_children)//2)): # mutate half of the populat
            # select random individual
            child = np.random.randint(0, len(x_children))

            # select random mutation point
            mutation_point = np.random.randint(0, len(x_children[i]))

            # select random mutation value
            mutation_value = np.random.randint(self.bounds_min[mutation_point

            # mutate
            x_children[child][mutation_point] = dict[mutation_point+1][mutati
            # print("Mutation: " + str(child) + " " + str(mutation_point) + "

        return x_children

    def survivor_selection(self, f_children):
        """
        Survivor selection
        Sort the population by fitness
        The 10 best will be selected as parents
        """
        # sort the population by fitness
```

```
        f_children.sort()

        # select the best individuals
        f = f_children[:self.pop_size]

        # print the best fitness value of that generation
        print("Best Fitness (gen): " + str(f[0][0]))

        # select the configuration of the best individuals
        for i in range(len(f)):
            f[i] = f[i][2]

        return f

    def evaluate(self, x_children):
        """
        Run the network which each configuration
        """
        net = self.network(x_children)
        result = net.run()
        return result

    def step(self, x_old):

        x_parents = self.parent_selection(x_old) # select parents
        x_children = self.recombination(x_parents) # recombination
        x_children = self.mutation(x_children) # mutation
        f_children = self.evaluate(x_children) # evaluate the children
        x_new = self.survivor_selection(f_children) # survivor selection

        return x_new , f_children
```

In [ ]:

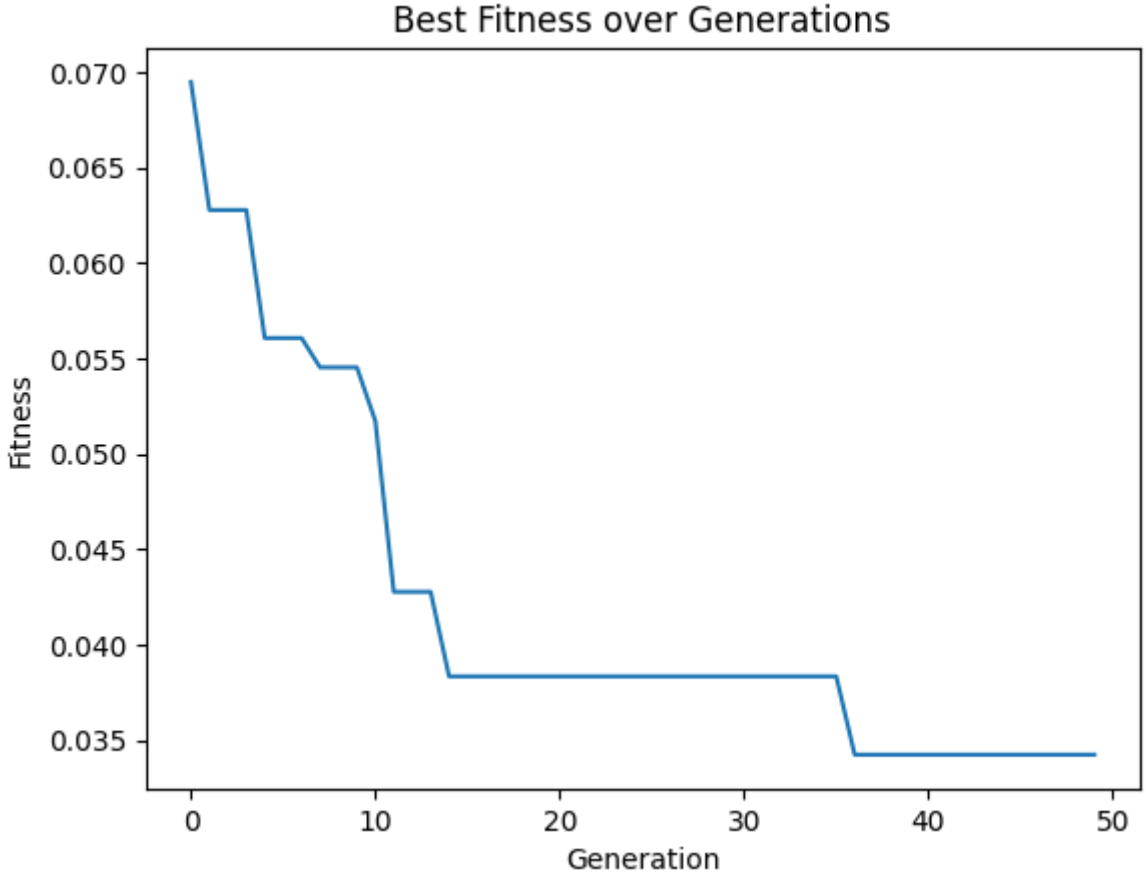
```
num_generations = 50

# initialize the evolutionary algorithm
ea = EvolutionaryAlgorithm(Network, population)

# run the evolutionary algorithm
best = None # initialize the best fitness value
best_list = [] # initialize the list of overall best fitness values

for generation in range(num_generations):
    print("Generation: ", generation+1)
    population, f_and_c = ea.step(population)

    # print(f_and_c[0][0])

    if best == None:
        # if best is None, initialise first fitness value
        best = f_and_c[0]
        best_list.append(best[0])
    elif best[0] > f_and_c[0][0]:
        # if new fitness value is better than old one, replace it
        best = f_and_c[0]
        best_list.append(best[0])
    else:
        # if new fitness value is worse than old one, re-add old one
        best_list.append(best[0])

    # print the best fitness value of that generation
    print("BEST FITNESS: " + str(best[0]))

    # save the overall best fitness value and config of that generation
    f = open(result_dir + "best.txt", "w")
```

```
    f.write("Fitness: " + str(best[0]) + "\nConfiguration: " + str(best[2]))
    f.close()


print("FINISHED!")

print("Best Configuration: " + str(best[2]))

# plot the best fitness over num_generations
plt.plot(best_list)
plt.xlabel("Generation")
plt.ylabel("Fitness")
plt.title("Best Fitness over Generations")
plt.savefig(result_dir + "best_fitness.png")
plt.show()
```

```
Generation:  1
Best Fitness (gen): 0.0694846019554863
BEST FITNESS: 0.0694846019554863
Generation:  2
Best Fitness (gen): 0.06277319255951314
BEST FITNESS: 0.06277319255951314
Generation:  3
Best Fitness (gen): 0.06277319255951314
BEST FITNESS: 0.06277319255951314
Generation:  4
Best Fitness (gen): 0.06277319255951314
BEST FITNESS: 0.06277319255951314
Generation:  5
Best Fitness (gen): 0.05606178316353997
BEST FITNESS: 0.05606178316353997
Generation:  6
Best Fitness (gen): 0.05614237903825349
BEST FITNESS: 0.05606178316353997
Generation:  7
Best Fitness (gen): 0.06099696773157097
BEST FITNESS: 0.05606178316353997
Generation:  8
Best Fitness (gen): 0.05452664567962633
BEST FITNESS: 0.05452664567962633
Generation:  9
Best Fitness (gen): 0.056522694800922195
BEST FITNESS: 0.05452664567962633
Generation:  10
Best Fitness (gen): 0.056522694800922195
BEST FITNESS: 0.05452664567962633
Generation:  11
Best Fitness (gen): 0.051716107126193976
BEST FITNESS: 0.051716107126193976
Generation:  12
Best Fitness (gen): 0.04275941255086538
BEST FITNESS: 0.04275941255086538
Generation:  13
Best Fitness (gen): 0.04964512803290932
BEST FITNESS: 0.04275941255086538
Generation:  14
Best Fitness (gen): 0.058751682552215516
BEST FITNESS: 0.04275941255086538
Generation:  15
Best Fitness (gen): 0.03832995754738747
BEST FITNESS: 0.03832995754738747
Generation:  16
Best Fitness (gen): 0.04947082194683853
BEST FITNESS: 0.03832995754738747
Generation:  17
Best Fitness (gen): 0.04263489001457813
BEST FITNESS: 0.03832995754738747
Generation:  18
```

```
Best Fitness (gen): 0.04980313669091798
BEST FITNESS: 0.03832995754738747
Generation:  19
Best Fitness (gen): 0.056348388714851406
BEST FITNESS: 0.03832995754738747
Generation:  20
Best Fitness (gen): 0.04748292153957373
BEST FITNESS: 0.03832995754738747
Generation:  21
Best Fitness (gen): 0.04980313669091798
BEST FITNESS: 0.03832995754738747
Generation:  22
Best Fitness (gen): 0.054443566993606474
BEST FITNESS: 0.03832995754738747
Generation:  23
Best Fitness (gen): 0.047566000225593597
BEST FITNESS: 0.03832995754738747
Generation:  24
Best Fitness (gen): 0.049557974989873924
BEST FITNESS: 0.03832995754738747
Generation:  25
Best Fitness (gen): 0.04955390063285839
BEST FITNESS: 0.03832995754738747
Generation:  26
Best Fitness (gen): 0.0519571944702225
BEST FITNESS: 0.03832995754738747
Generation:  27
Best Fitness (gen): 0.05419433093554688
BEST FITNESS: 0.03832995754738747
Generation:  28
Best Fitness (gen): 0.0519571944702225
BEST FITNESS: 0.03832995754738747
Generation:  29
Best Fitness (gen): 0.045245785074249346
BEST FITNESS: 0.03832995754738747
Generation:  30
Best Fitness (gen): 0.049349928134768246
BEST FITNESS: 0.03832995754738747
Generation:  31
Best Fitness (gen): 0.047649078911613456
BEST FITNESS: 0.03832995754738747
Generation:  32
Best Fitness (gen): 0.052123351842262224
BEST FITNESS: 0.03832995754738747
Generation:  33
Best Fitness (gen): 0.052123351842262224
BEST FITNESS: 0.03832995754738747
Generation:  34
Best Fitness (gen): 0.049720058004898114
BEST FITNESS: 0.03832995754738747
Generation:  35
Best Fitness (gen): 0.049720058004898114
BEST FITNESS: 0.03832995754738747
Generation:  36
Best Fitness (gen): 0.05419433093554688
BEST FITNESS: 0.03832995754738747
Generation:  37
Best Fitness (gen): 0.034226260119667146
BEST FITNESS: 0.034226260119667146
Generation:  38
Best Fitness (gen): 0.05187411578420264
BEST FITNESS: 0.034226260119667146
Generation:  39
Best Fitness (gen): 0.05187411578420264
BEST FITNESS: 0.034226260119667146
Generation:  40
Best Fitness (gen): 0.04739984285355387
BEST FITNESS: 0.034226260119667146
Generation:  41
```

```
Best Fitness (gen): 0.049636979318878255
BEST FITNESS: 0.034226260119667146
Generation:  42
Best Fitness (gen): 0.04516270638822949
BEST FITNESS: 0.034226260119667146
Generation:  43
Best Fitness (gen): 0.049636979318878255
BEST FITNESS: 0.034226260119667146
Generation:  44
Best Fitness (gen): 0.05187411578420264
BEST FITNESS: 0.034226260119667146
Generation:  45
Best Fitness (gen): 0.04068843345758071
BEST FITNESS: 0.034226260119667146
Generation:  46
Best Fitness (gen): 0.056348388714851406
BEST FITNESS: 0.034226260119667146
Generation:  47
Best Fitness (gen): 0.047649078911613456
BEST FITNESS: 0.034226260119667146
Generation:  48
Best Fitness (gen): 0.047649078911613456
BEST FITNESS: 0.034226260119667146
Generation:  49
Best Fitness (gen): 0.04267633386484551
BEST FITNESS: 0.034226260119667146
Generation:  50
Best Fitness (gen): 0.05858552518017579
BEST FITNESS: 0.034226260119667146
FINISHED!
Best Configuration: [32 (3, 1) <class 'torch.nn.modules.activation.Tanh'> 2
 <class 'torch.nn.modules.pooling.MaxPool2d'> 80]
```



In [ ]:

In [ ]: