Queen Mary
University of London

ECS759P Artificial Intelligence

Coursework 2

Manuela Valencia Toro

240429850

Genetic Algorithm

Best model:

Architecture:

nn.Linear(in_features=3334, out_features=271, bias=True), nn.ReLU(), nn.Linear(in_features=271, out_features=6, bias=True), nn.Dropout(0.10227016708579334)]

Accuracy:

0.4318

1.2 Elements of the algorithm

State encoding

To define each individual the algorithm creates a list containing the population which is represented by lists of dictionaries. The size of the population defined is the number of lists to return. Each single list can be defined as a genome that finally would be converted into an architecture. Each genome has a number of layers, that are represented with dictionaries. Each one of those dictionaries contains the number of neurons, the activation function and the dropout. this is an example of a genome's structure:

[{'num_neurons': 182, 'activation': 'softmax', 'dropout_rate': None}, {'num_neurons': 182, 'activation': 'softmax', 'dropout_rate': 0.2}]

Initial Population:

For each architecture to create based on the parameter Size, the algorithm generates a number between 1 and max_n_layers to define the number of layers ensuring all the architectures have enough variance and have multiple performances to increase the possibility of finding an optimal combination. Each layer is assigned a random number of neurons, activation, and dropout values. To do it, since they are lists, the algorithm generates a random index in each case and gets its value from the respective parameter, creates the dictionary and finally appends it to the population list.

Selection

In order to implement the selection process, as the individuals with higher fitness have to be chosen, for each parent to return defined in the num_ parents parameter, the algorithm finds the index of the maximum value in the fitnesses list that basically is the best performance, gets the value on that index from the parameter population and assigns it to a variable sorted_population, and remove both values in both lists, fitnesses and populations, to avoid taking them again in the next iteration. Finally, it returns a list with n_parents individuals.

Crossover

Basically, the crossover process involves combining 2 parent genomes to create a new child genome. To implement it, since both parents have high accuracy, the algorithm creates the new child with the mean between the layers of the parents. After that, It chooses randomly between activation, neurons, and drop out of the two parents by creating a list with both values, like this:

[Activation from parent 1, Activation from parent 2]

In order to select a configuration it generates an index randomly and takes its value to finally assign it to the children's genome. All the values are selected randomly and because of that it is possible to create multiple combinations that could improve the performance.

Mutation:

The main idea of this process is to add some little changes to the genome in order to improve its performance. Some of these little changes were applied as noise drawn from normal distribution based on a defined sigma to control the variability and a learning rate to control the magnitude of the mutation. The random noise is calculated as sigma prime by this way.

$$\sigma' = \sigma \cdot \exp\big(\tau \cdot \mathrm{N}(0,1)\big)$$
$$x_i' = x_i + \sigma' \cdot \mathrm{N_i}(0,1)$$

In this case, there are 2 different values for sigma, 5 and 0.1 to generate different ranges to use, and a learning rate of 0.1 used in both.

The number of layers continues the same avoiding loss of important characteristics of the architecture to mutate. For each layer, the new number of neurons is updated by adding the generated noise to the actual number of neurons in order to increase or decrease them, here the highest sigma was used to ensure the range of the possible changes would impact the performance. Since the activation could not be changed in terms of noise, the algorithm generates a random binary number, if it is 1 the activation is updated, else it is assigned

without any mutation. Finally, the dropout is also updated by using the normal distribution with 0.1 as sigma, ensuring most of the generated values are lower than 1, which is the maximum number the dropout could have.

1.3 The number of reproductions

Based on what I observed while I was executing the code, for any 10 times it is executed, it converges to the reported architecture around 5 or 6 times. The accuracy of these architectures had values between 38% and 41%, a similar performance to the optimal.
These are some examples of architectures with the highest performance:
[{'num_neurons': 121, 'activation': 'relu', 'dropout': 0.1}]
[{'num_neurons': 128, 'activation': 'relu', 'dropout': None}]
[{'num_neurons': 291, 'activation': 'elu', 'dropout': 0.2}]

From this, as it is easy to see they follow a pattern, I can conclude they are related to the reported architecture. In general, these architectures have only one or two single hidden layers, mainly relu, elu, and sigmoid as activation functions and variations in the number of neurons and dropout. If I ensure the initial population is mostly composed of architectures with a low number of layers, the probability of having higher performances will increase.

1.4 Hyper-parameters

Population size	**Best performance(%)**
13	**41**
12	**38**
9	**39**
8	**40**
7	**38**
6	**36**
5	**16**
4	**33**
3	**17**

The hyper-parameter population size defines the number of individuals for each generation. As can be seen in the table above, obtained by running the process with multiple values for population size, the larger it is, the greater the final accuracy tends to be and the higher the probability of convergence to an optimal architecture. This is because more randomly generated architectures are more likely to produce optimal combinations of layers, neurons, activation functions and dropouts in the first generation, ensuring greater variation between individuals than, for example, a high number of generations. If an architecture performs well in the first generation, the likelihood that it will be improved by the mutation methods is high, as only small changes need to be made to adjust the parameters, whereas an architecture that performs poorly from the start will require more complex changes and iterations to eventually achieve good performance. Although increasing the population might improve performance, it also increases the number of iterations. Therefore, it is important to find a good balance between them. I used 6 as the population size and it performs well with a good response time.

Lost in the closet (Classification)

2.1 Most appropriate loss function to use

Cross-Entropy Loss

In general, this loss function measures the difference between the predicted probability distribution and the true distribution of classes and helps the model to make more accurate predictions. I chose Cross-entropy since each image can belong only to one possible class and the returned classification is a matrix of probabilities without softmax conversion.

In PyTorch, the cross-entropy function is provided by nn.CrossEntropyLoss(). As the model returns the probability of each image belonging to each class, this function applies softmax to make sure the sum of all the probabilities for each image is one, and by getting the class with the highest probability compares it to the target, that basically is a list of 10 numbers indicating the class of each image. This is the formula:

$$\mathcal{L} = -\sum_{c=1}^{C} y_c \log(p_c)$$

Where,

- **C** is the number of classes, that in this specific case is 10.
- **Yc** is the true label
- **Pc** is the predicted probability for class C and is obtained the softmax function calculated by this way:

  $$Pc = \frac{exp(Zc)}{\sum_{i=1}^{C} exp(Zi)}$$

  Here, **Zc** is the raw output for class C. Basically it works by normalizing the values.

Here, the Logarithmic function penalizes high differences making the loss large if the predicted probability for the correct class is small, this is, the probability for the correct class has a number significantly lower than 1, and small if it is closer to 1 meaning a classification near to the ground truth. The objective is to maximize it making the predicted probability for the correct class as high as possible.

Gradient with respect to the Class Scores

The gradient descent is the derivative of the loss function and is used to minimize the total loss by updating the model parameters based on the class scores.

To compute this derivation the chain rule is applied to finally get this formula:

$$\frac{\partial \mathcal{L}}{\partial z_c} = p_c - y_c$$

As you can see the gradient descent is given by the difference between the true value and the predicted probability of class C. In this way, if Pc and Yc are the same, this means, the image was correctly classified, and the respective weight is not going to change, but if there is a difference between them, the greater the difference, the greater the change in the weight.

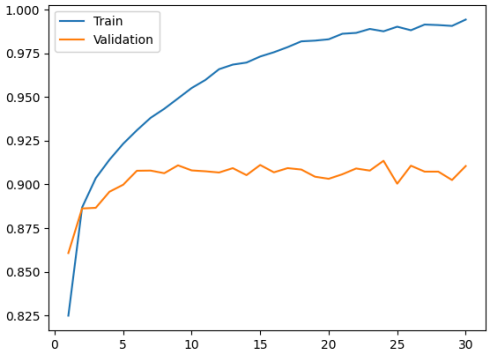
2.2

a) final train and validation accuracy obtained

Train Accuracy : 0.9943333333333333

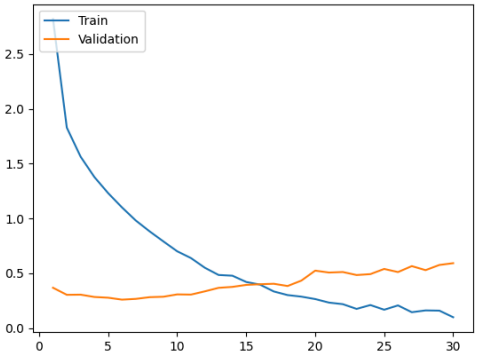
Evaluation Accuracy: 0.9105

b) plot of the accuracy on the training and validation sets per each epoch, comment on the speed of performance changes across epochs.



This diagram shows the accuracy (y-axis) per epoch (x-axis) for the training and test set. It is easy to see that the accuracy increases with the number of epochs. The speed of this increase is higher for both sets around the first epochs, from 1 to about 6, because the model starts to learn simple patterns of the data. From this point the speed reduces and there is a notable difference in performance between the training set and the test set, at this point the values for training are higher, which probably shows that the model is overfitted.

c) plot of the train and validation loss per epoch and comment on the speed of decrease.



This graph shows the loss per epoch for the training and test set. In the first epochs, the speed of the decrease is higher for the training set because, as previously described, the model starts

to learn patterns from the data and adjust the performance until it stabilizes, but in general it shows a decreasing behavior, which means that the model converges to a minimum. Therefore, the behavior of the test set only shows a decrease in the first 2 epochs, after that there is an increasing behavior, which is not normal in the loss function and shows that the model is overfitting.

Planning Logic

3.1 Listing pre-conditions and effects associated with actions.

1. Take whole Apple:

    Preconditions:

    - Should be available

    Effects:

    - The apple is in the robotic assistant’s hand
    - There is not an available apple

2. Peel Apple

    Preconditions:

    - Apple should be clean
    - Apple shouldn't have been peeled before
    - Apple should be in the robotic assistant’s hand
    - Knife should be in the robotic assistant’s hand

    Effects:

    - The apple is peeled

3. Cut Apple

    Preconditions:

    - Apple should be peeled
    - Apple shouldn't have been cut before
    - Apple should be in the robotic assistant’s hand
    - Knife should be in the robotic assistant’s hand

    Effects:

    - The apple is cut

4. Put down Knife

Preconditions:

- Knife is in robotic assistant's hand

Effects:

- The knife is not in the robotic assistant's hand

5. Add chopped apples to the container (in order to serve to your guests).

Preconditions:

- Apple should be in the robotic assistant's hand
- Container is empty

Effects:

- Container contains the apple
- Container is not empty
- Apple is not in the robotic assistant's hand

3.2 Simple Statements with First-order logic

1. Take whole Apple

Available(Apple)

2. Peel Apple

is_clean(apple) ^ ¬ is_peeled(apple) ^ in_hand(apple) ^ in_hand(knife)

3. Cut Apple

is_peeled(apple) ^ ¬ is_cut(apple) ^ in_hand(apple) ^ in_hand(knife)

4. Put down Knife

in_hand(knife)

5. Add chopped apple to the container (in order to serve to your guests).

in_hand(apple) ^ empty(container)

3.3 FOL to CNF

1. The action of peeling an apple.

FOL with implies connective:

is_clean(apple) ^ ¬ is_peeled(apple) ^ in_hand(apple) ^ in_hand(knife) => is_peeled(apple)

CNF

P => Q → ¬P v Q

- *¬ (is_clean(apple) ^ ¬ is_peeled(apple) ^ in_hand(apple) ^ in_hand(knife)) v is_peeled(apple)*

¬(P ^ Q) → ¬P v¬ Q

- *¬is_clean(apple) v is_peeled(apple) v ¬ in_hand(apple) v ¬ in_hand(knife) v is_peeled(apple)*

Repeated predicate is removed

- *¬is_clean(apple) v is_peeled(apple) v ¬ in_hand(apple) v ¬ in_hand(knife)*

2. The action of cutting an apple.

FOL with implies connective:

is_peeled(apple) ^ ¬ is_cut(apple) ^ in_hand(apple) ^ in_hand(knife) => is_cut(apple)

CNF

P => Q → ¬P v Q

- *¬(is_peeled(apple) ^ ¬ is_cut(apple) ^ in_hand(apple) ^ in_hand(knife)) v is_cut(apple)*

¬(P ^ Q) → ¬P v¬ Q

- *¬is_peeled(apple) v is_cut(apple) v ¬ in_hand(apple) v ¬ in_hand(knife) v is_cut(apple)*

Repeated predicate is removed

- *¬is_peeled(apple) v is_cut(apple) v ¬ in_hand(apple) v ¬ in_hand(knife)*

3.3 Linear Temporal Logic

Initial Conditions

G(

available(apple) ∧ available(knife) ∧ available(container) ∧ is_clean(apple) ∧ ¬ is_peeled(apple) ∧ ¬ is_cut(apple) ∧ empty(container)

)

Sequence of Actions

```
G(
        take(apple) →
        X (
                take(knife) →
                X (
                        peel(apple,knife) →
                        X (
                                cut(apple,knife) →
                                X (
                                        take(container) → X (add_to(apple,container))
                                )
                        )
                )
        )
)
```