Neuroevolution – Deliverable 2

Filipe Marçal Dias (r20181050), Inês Santos (r20191184), Manuel Marreiros (r20191223)

Grammar, Genotype and Phenotype

We started our project by defining the **grammar**, which consists of a dictionary, *layer_params*, that contains all the possible layers and respective parameters that can be used by our models.

We then have a function called *generate_random_network,* which takes as inputs the *input_size* (hardcoded to 28x28 – the dimensions of the MNIST dataset we used to train the networks), the *output_size* (hardcoded to 10, since these are the number of labels in the dataset), *min_layers* (2) and *max_layers* (50). This function randomly selects the number of layers, *num_layers,* within the given range, and then it goes through the dictionary and draws that number of random layers with parameter values randomly selected from the different possibilities we defined. All the layers are then added to a *layer*_list, which stores the layers and its configurations. This *layer_list* represents the randomly generated neural network architecture and is ultimately returned in the end.

While the intermediate layers are all generated at random, the first and last ones are pre-established from the get-go: we begin with a linear layer that takes 512 features, and we end with another linear layer with an output size of 10. We opted to include the *SoftMax* layer in the training stage and only convert the raw output of the model into probabilities at that point.

Another constraint that we added manually was that the layers which required the number of input features to be specified had to use the number of features that was outputted by the last layer. We controlled this situation by using a variable called *previous_output_features*, which kept track of the current number of features at each stage. This variable was only modified once we added a *Linear* layer, as that is the only type of layer with the capability to alter the number of features.

Moving on, we then define a network class (*Net*) to parse instructions and build the *PyTorch* network structure, the **phenotype**, with the information generated by the previous function. The core method inside this class is the *parse_layer_string*, which iterates over each layer string in *layer_list* and returns the corresponding *PyTorch* layer module based on the layer type specified in the string.

Regarding the **optimizers**, similarly to what was done with the layers, we begin with a dictionary, *optimizer_params*, which contains 5 types of optimizers, the respective parameters they accept, and some values for them.

We then have a *build_optimizer* function, used to parse the optimizer instructions and construct the corresponding *PyTorch* optimizer. The function takes three parameters: *optimizer_str* represents the optimizer type specified as a string, *params* is a dictionary containing the optimizer parameters, and *model* is the PyTorch model for which the optimizer will be created. The function follows a series of conditional statements to determine the optimizer type and its associated parameters. For example, if the optimizer string is "SGD", it extracts the learning rate (*lr*), *momentum*, and *Nesterov* flag from the *params* dictionary and creates an instance of *torch.optim.SGD* with the specified parameters. Similarly, other optimizer types such as "Adam", "AdamW", "NAdam", and "Adadelta" are supported with their respective parameter extraction and optimizer instantiation. Finally, the function returns the constructed optimizer.

Network creation

The creation of the network was quite simple, and it was done in the exact same way for the five networks. First, we used the aforementioned *generate_random_network* function to get a list of randomly generated layers in a string format. Then, with that list, we created an instance of the *Net* class, and we defined the loss function. We then randomly selected an optimizer and some of its parameters, using the function *build_optimizer* to construct it. Once this was done, we utilized a function that trained the model and allowed us to discern the loss and the accuracy for both training and validation as epochs evolved. For computational purposes, we opted to use only 10 epochs instead of the 50 that are recommend in the guidelines, as such a high value was taking too long to run.

Genetic Operators

Crossover

We first performed the **crossover** between networks 1 and 2. To do that we used a function that begins by creating copies of *network1* and *network2* to avoid modifying the original networks. This is important because crossover involves swapping layers between the networks, and we wanted to preserve the integrity of their original form.

Next, the function determines the cutoff points for crossover. The cutoff points are selected within the valid range, which is determined by the smaller network's size. This ensures that the crossover operation can be performed without exceeding the boundaries of either network and that neither the first nor the last layers are affected. Two unique cutoff points are randomly chosen, and they are sorted in ascending order.

The function then proceeds to swap the blocks of layers between the networks. Starting from the first cutoff point and ending at the second cutoff point, including them both, the layers at the corresponding positions in *network1_copy* and *network2_copy* are swapped.

After the swapping is done, it's important to ensure that the **input features of a given layer are still coherent with the output of the previous one**. This can be easily disrupted when doing the crossover operation, so, to prevent this from happening, we added a mechanism that adjusts the feature values in the swapped layers, specifically for linear layers and normalization layers (*BatchNorm1d* or *LayerNorm*).

Finally, the function returns the modified copies of the networks, *network1_copy* and *network2_copy*, representing the result of the crossover operation.

Mutation

Regarding the **add layer mutation**, we simply pick a random index, generate a layer, and insert it. To generate a random layer, we go back to the *generate_random_network*, which will return a network with at least three layers. However, since the first and last layers are default, we only care for the middle one, which is going to be inserted the network.

Similar to what was done in the crossover operator, we needed to ensure the newly added layer was coherent with the ones that were already there. In that sense, we added an if statement that first checks if the new layer belongs to a type that requires the number of features to be specified. If so, it searches for the last linear layer before the insertion position and retrieves its output size. Then, depending on the type of the new layer, appropriate modifications are made. For Linear layers, both the *in_features* and *out_features* attributes are set to the output size of the last linear layer, preserving the sequence. For *BatchNorm1d* layers, the *num_features* attribute is updated to match the output size. Regarding *LayerNorm* layers, the *normalized_shape* attribute is also adjusted for consistency. Ultimately, the adjusted new layer is inserted into the network at the desired

position. This ensures that the number of features in the new layer aligns with the preceding linear layer, maintaining the desired property of the layer sequence.

To **remove a layer**, the same logic is followed. However, once the index is determined, we simply remove that layer. Like before, we need to ensure the consistency between input/output values of layers, so the code checks if the removed layer is a linear layer. If it is, it searches for the index of the next linear layer in the genotype and stores it in *next_linear_index*. If a valid next linear layer index is found, it accesses the corresponding layer object, *next_linear_layer*, and updates its *in_features* attribute to match the input size of the removed layer. Then, it iterates over the layers between the removed layer and the next linear layer (exclusive), and if a layer is a *BatchNorm1d*, it updates its *num_features* attribute to match the input size of the removed layer. Similarly, if a layer is a *LayerNorm*, it updates its *normalized_shape* attribute to have 16 as the batch size and the input size of the removed layer. This ensures that the necessary layer parameters are updated and maintains consistency until the next linear layer in the network genotype.

Finally, for the **optimizer change mutation**, we defined a function that takes a model and an optimizer string as input. The function randomly selects a new optimizer from the predefined list of optimizer options. If the new optimizer is the same as the old one, it randomly selects a parameter and changes its value to a new random value from the parameter's possible values. If the new optimizer is different, it randomly selects new values for all the parameters of the new optimizer. The function then prints the newly generated optimizer string and its parameters. Finally, it rebuilds the optimizer using the new optimizer string and parameters and returns the optimizer. This function allows for mutating the optimizer by changing its type or parameter values while ensuring the generated optimizer remains valid.

Once we applied the crossover and mutation operators, we retrained the networks to see how they compared to the original ones. Analysing the results, we concluded that the crossover operator did not yield great results in any of the newly generated networks, even decreasing the accuracy of the first one. The same can be said for the different mutations, which did not have a great impact in terms of performance. Although disappointing, this was expected, since none of them made major changes to the architecture of our networks.