1. Defining a string-based representation for the networks (the genotype).

The "dict_network" variable represents a grammar that defines possible configurations for building a neural network, that allows for flexibility in the choice of number of features, activation functions and dropout and normalization layers. To ensure that a function Pytorch neural network can be built based upon this grammar, the structure was defined so that the selection of the layers is not completely random but within a given framework.

First, "Linear" represents a fully connected layer in the network. The keys "in_features" and "out_features" contain the number of possible values for the input and output features, respectively, from the list [128,256,512]. The "bias" key is a list of Boolean values indicating whether a bias is added to the fully connected dense layer or not.

For the normalization methods another nested dictionary was defined, containing the normalization methods "BatchNorm1d" and "LayerNorm". The Batch Normalization layer for 1D data contains "eps", the possible values for the epsilon parameter, being [0.1, 0.01, 0.001], and "momentum", that includes the possible values for the momentum parameter [0.1,0.5,0.9]. Layer Normalization can take in different values of "eps" being [1e-5, 1e-4, 1e-3].

There are two possibilities for the dropout layer, represented by within a nested dictionary with the key "drop", that can be "Dropout" and "AlphaDropout", respectively, the standard Dropout layer and the Alpha Dropout layer. Both dropout possibilities hold the possible values of probabilities as [0.1, 0.3, 0.5, 0.7]. Lastly, the "activation" key contains a list of activation functions, such as "Sigmoid", "ReLU", "PReLU", "ELU", "SELU", "GELU", "CELU", and "SiLU". These functions introduce non-linearity to the network, enabling it to learn more complex patterns.

The function 'dict_to_string_representation_network' will use the 'build_block' function to generate a string representation of a neural network architecture. It takes in as parameters the total input size, total output size, network configuration dictionary, minimum and maximum amount of layers, dropout layer and normalization layer rate, which have the default value of 0.5. In the beginning, the total input size and total output size are converted to strings and an empty list to store the string representation of the entire network is initialized. It then generates a number of layers between the minimum and maxim amount that will determine the number of layers to add in the network. A loop is created until this number is achieved and, inside the loop, the function 'build_block' is called and iterated over until each layer is appended to the 'total_string_representation' list. Within the loop also the information regarding the input/output features is saved, so that the blocks can be built in a functional manner.

The function 'build_block' takes in several parameters: 'dict_network'; 'in_features', the number of input features; 'out_features', the number of output features; 'drop_layer_rate', the dropout layer rate; 'norm_layer_rate', the normalization layer rate and, lastly, 'out_features_fix', a boolean value indicating whether the number of output features is fixed or randomly chosen. The function initializes an empty list to store the string representation of each layer in the block. Next the layer-block is built. It was decided that each block starts with a linear layer, so that a functional PyTorch model can be created. Following, the block is built starting by adding a linear layer, adding the parameters received from the 'dict_to_string_representation_network' and adding them together to build the genotype.

The function then continues by choosing random values from the number of input and output features, and from the bias. If "random.uniform(0,1)" is lower than the probabilities of dropout introduced in the parameters of the function, one of two dropout layers will be added on which its values will be randomly chosen from "dict_network" and the same procedure is applied to the normalization layer. In the last step, this function adds an activation function to the block by appending a string representation starting with "Act|" and a randomly chosen activation function. Finally, the function returns the "total_string_list" and the value of "out_features". If the block represents the last layer, the output nodes are not chosen randomly but are the number of classes (10).

2. Defining a network class that can parse those instructions and build a functional pytorch network structure (the phenotype)

To build the phenotype, the class 'nn.Module' was created, that allows for the creation of a neural network based on the genotype. After the class is intialised a PyTorch Module List is defined which can hold the submodules of PyTorch neural networks like a regular python list. It is then iterated through the provided genotype, utilizing the 'parse_layer_string' function to put the layer information into PyTorch Neural Network classes. These classes are then appended to the module list and are returned as a PyTorch Neural Network Class.

Additionally, we created the method 'parse_layer_string' converts the string based layers into PyTorch layer objects. There is a loop iterating through all the existing layers - Linear, Batch Normalization, Layer Normalization, Standard Dropout, Alpha Dropout and Activation Function; that checks the type based on the prefix of the string and extracts the relevant parameters that were randomly chosen previously according to 'dict_network'. If the layer string doesn't match any of the types mentioned previously, it raises a ValueError with an appropriate error message.

3. Defining a string-based representation for the optimizer.

The dictionary 'dict_optimizer' contains several optimization algorithms, corresponding to the keys, and its respective hyperparameters, corresponding to the subkeys. The optimization algorithms are Adam, that contains the learning rate ("lr") and "betas" hyperparameters; AdamW, that, in addition to the "lr" and "betas", it also has the "weight_decay" hyperparameter; Adadelta contains the "lr" and "rho" hyperparameters; NAdam, with the "lr", "betas" and, additionally, it also has the momentum decay ("mom_decay"); and SGD, that has the learning rate ("lr"), "momentum", and the Boolean "nesterov" hyperparameter.

The function 'dict_to_string_representation_optimizer' takes the previous dictionary as an input and randomly selects an optimization algorithm from the five available. It then randomly selects values for each hyperparameter of the chosen optimization algorithm and returns it as a string representation.

4. Defining a way to parse those instructions and build a functional pytorch optimizer.

The function "parse_opt_string" takes two parameters, the string representation of the optimizer and the model for which it should be built. First, the parameters of the model are retrieved using the PyTorch parameters() method. Then, the optimizer type is checked with the startswith() function, looking at the first part of the string representation. Next, the string is split by the semicolons within. This split is the used to retrieve the hyperparameters which are the passed to the Py Torch "optim" class for the respective optimizer. By adding the parameters of the model to the class as well, it is ensured that a functional Py Torch optimizer is built.

5.1. Network Crossover

The defined crossover takes a subset of layers from two different networks and swaps them. However, the subset cannot include the first or the final layers. The name of the function is "crossover" and takes the two parent layers as input.

First, the length of the layers is assessed, and an error is thrown if they are too small for the genetic operation. Next, the two cut off points are chosen randomly within the smaller network. This was implemented to ensure that the layer-subset doesn't exceed the size of one of the parents. Furthermore, to make sure that the size of the crossovers is bigger than zero, the points are chosen over and over until they do not have the same value. Finally, the crossover is conducted. The subsets are exchanged using slicing.

In the next step the networks need to be rebuilt to be functional. The reason for this is that the input and output parameters of the inserted subsets are not matched to the respective networks, which would eventually break them. For this process we defined the function "crossover_adaptation" which takes the respective offspring, and both cut off points as inputs.

The function works as follows: First, it looks for the first layer containing an input parameter (Linear, BatchNormID or LayerNorm) before cut off point one. To do so it subtracts one of the cut off position until a corresponding layer is found. Next, the output parameter of the found layer is retrieved and saved. The same procedure is then done for the first layer containing input parameters after the second cut off point and saving its value. With the contained information the layer subset can be adapted to the network. To do so we iterate through the block, layer by layer and adapt the input values if needed. There are three different scenarios where parameters need to be changed:

First, the layer is a liner layer. If that is the case the input parameters are adapted to the output parameters of the network, which were saved before. On top of that the variable containing the output parameters are overwritten with the output parameters of the layer, to ensure that this parameter change is not lost.

Second, the layer is a batch normalization or a layer normalization layer. In this case the input/output parameters get adapted to the output saved which can either be the one from the second parent network or from a preceding linear layer as described above.

Third, the layer is the last linear layer in the subset. If that is the case the input is changed to the output saved and the output is changed to the next input of the retrieving parent, which was saved before. The position of the last linear layer was also detected by iterating through the subset before the adaption loop.

This is done for both offspring networks ensuring that they can be built to be a functional PyTorch neural network.

5.2 Add layer mutation

This mutation adds a layer to any part of the genotype and the rebuilds the network. The only restriction is that it cannot be added before the first or after the last layer.

The mutation is defined in the function "add_layer_mutation" and it takes the grammar defined to build the network and a genotype network as inputs. First, the genotype is checked to assess if it is long enough to add a layer. If it is not an error is generated, asking the user to generate another genotype. If the network is large enough, another random genotype is generated using the given grammar and the function "dict_to_string_representation_network". Out of this "mutation-genotype" one random layer is chosen to put into the network. Next the position where the layer should be put is chosen randomly and the type of the "input layer" is saved.

The original layer is inserted into the network at the chosen position. Now the network needs to be rebuilt in order to be functional. First, the type of the new layer is checked. If it is a Dropout or an alpha dropout layer, no action is needed since there are no input or output parameters for these layers.

If the layer is linear the input and output parameters need to be adapted. The same procedure as described in the crossover is used to adapt the layer. The type and input/output parameters of the last relevant layer before and after the input layer are retrieved and saved and the input and output parameters of the new layer are adapted accordingly. If the mutation layer is of the type "BatchNorm1d" or "LayerNorm" the output of the last relevant layer before is used to adapt the layer.

5.3 Remove layer mutation

For this mutation any layer that is neither the first nor the last one should be removed from the genotype and then the network should be rebuild. For this mutation another restriction was implemented, which is that no linear layer can be removed. This was implemented to ensure that the network does not have to be rebuild and is still functional. The mutation was defined in the function "mutation_remove_layer" which takes one genotype network as input.

Once again, the function checks if the network is large enough for the mutation. If that is the case, it is ensured that no linear layer is removed for the explained reason. Then a copy of the network is made and using the "pop" method, a randomly chosen layer is removed from the offspring.

5.4 Change Optimizer Mutation

The change optimizer mutation should change any parameter from the optimizer genotype and rebuild it. If the type of the optimizer is changed the parameters must also be changed to ensure a valid optimizer is generated. The function for this mutation is called "mutation_change_opt" and takes the optimizer genotype and the optimizer-grammar as input.

First, the genotype is split and saved into a parameters list. Then the parameter which should be changed is chosen randomly. Following, it is checked which parameter was chosen. If it was the optimizer type, a new optimizer is built from the grammar, utilizing the "dict_to_string_representation_optimizer" function described earlier. If the chosen parameter is not the optimizer type, the parameter options are retrieved from the grammar and then randomly chosen until it is different from the original optimizer. Finally, the new parameter value is input into the string and the genotype is rebuilt.