Neuroevolution Deliverable 2

Group A

Adriana Monteiro 20220604 Farina Pontejos 20210649 Marta Dinis 20220611
Patrícia Morais 20220638 Samuel Santos 20220609

June 11, 2023

1 Grammar

Our network definition is based on a grammar that is implemented as a dictionary with two top-level elements, "Layers" and "Optimizer".

The "Layers" entry stores a dictionary, in which each entry is a layer definition, whose key is the Layer Name, and whose value is a dictionary with parameters information. Inside each Layer's parameter dictionary, we have a key for each parameter that is relevant to that layer. The corresponding dictionary value will be a list with valid values for that parameter. The parameters and its values depend on the Layer.

The activation is implemented as a layer with one single parameter, `function_name`, whose list of values are the supported activation functions.

This approach is also implemented in the Optimizer top-level dictionary entry; its keys are optimizers, and the corresponding values are dictionaries that define the corresponding parameters domain:

```
grammar = { "Layers"      : {
              "Linear"          : { "n"             : [1,2,3,4,5,6,7,8,9,10],
                                    "bias"          : [True, False] },
              "Activation"      : { "function_name" : ["Sigmoid", "ReLU", ... ],
              "..."             : {...}                                          },
            "Optimizer"   : {
              "Adam"            : { "lr"    : [0.0001, 0.001, 0.01],
                                    "betas" : [(0.9, 0.99), (.95, .9)] },
              "Adadelta"        : { "lr"    : [0.01, 0.1, 1],
                                    "rho"   : [0.9] },
              "..."             : {...}                                          }}
```

2 Genotypes

2.1 Representation of the Genotypes

Genotypes are stored as strings that can also be represented as dictionaries. Transforming from a string to its corresponding dictionary is done with the function `parse_input`, mostly using regular expressions and string splits (on comma or on colon).

Layer parameters are stored in its string representation as a comma-separated sequence of key-value pairs, with a colon separating key from value. We pass this string to the `parse_args` function, which first splits it at the comma character, obtaining a list of key-value pairs. Each of these is further split at the colon character; the element to the left is considered a key and the element to the right is the value. The value is coerced into the appropriate datatype using the `recast` function, before actually being assigned as the value to its corresponding key.

The most significant difference between the genotype dictionary and the grammar dictionary is that each key in the genotype has only the actual value, not the list of possible values.

A neural network with one linear layer, followed by a BatchNorm1d Layer and another linear layer will look like this:

```
[ {"Linear"          : { "n" :4 , "bias" : True }            },
  {"BatchNorm1d"     : { "eps" :1e-5 , "momentum" :0.1 }     },
  {"Linear"          : { "n" :2 , "bias" : False }           } ]
```

2.2 Network

The network is essentially a list of layers that will be implemented sequentially. The first and last layers are necessarily linear layers. In fact, this constitutes the minimal neural network that can be created: just a linear input layer and a linear output layer. This is enforced when initializing a new random individual and in the add/remove layer mutations, discussed further in Section 3.2.

The rule that the network cannot have more than 50 layers is only implemented when creating a new network. Through both the add layer mutation and the cross-over, it is possible that a network ends up having more than 50 layers. Contrary to the previously mentioned rules, this does not prevent the network from running.

The consistency between the outputs of one level and the inputs at the next level has to be enforced in all situations, because otherwise the network would break.

2.3 The Optimizer

The representation of an optimizer uses the same logic that was described for a layer in the neural network, with the difference being that we can only have one optimizer in the model. Below is an example of an Optimizer representation.

```
{ "Optimizer": { "Adadelta" : { "lr" :0.01 , "rho" :0.9 } } }
```

2.4 From Genotype to Phenotype

The function `generate_network` is responsible for generating new random neural networks. Inside this function, we call `random_layers_generator`, which returns a list with between 2 and 50 layers, including the parameters for each of those layers. This list is checked for consistency, by calling function `enforce_layer_n` to enforce that both first and last layers are linear, and function `enforce_in_out_feats` to enforce that the number of output features at a layer is consistent with the number of output features at the next layer.

Once we have the list of layers, we instantiate the `Net` class, which is responsible for creating the PyTorch model represented by our dictionary. We also add the final `Softmax` output layer here.

The function `random_optimizer_generator` creates an optimizer genotype, as well as checks the additional restrictions for each optimizer. It enforces the specificities of the SGD optimizer: if parameter `"nesterov"` is defined and set to True, then the list of valid values for attribute `"momentum"` cannot include the first position in the list, which is `0`.

The `make_optimizer` function uses this optimizer genotype, as well as the `model.parameters()` obtained from constructing the PyTorch model, to create the PyTorch optimizer.

Finally, the `generate_network` function returns three objects: a dictionary representing the configuration of the model, the actual model, and the PyTorch optimizer.

3 The Genetic Operators

3.1 Crossover operator

We implemented the function `crossover` that takes two network genotypes as parameters and returns two mutated genotypes. For each parent genotype, we randomly choose two crossover points between the second layer and the penultimate layer. This divides the network in 3 parts, while ensuring that we keep the first and the last layer unchanged. We then switch the middle part between each network. Next we call function `enforce_in_out_feats` to ensure that the in features for each layer correspond to the out features from the previous one. Crossover could easily disrupt that. Performing this operation on a pair of networks in which one of the participants contains only two linear layers results in an empty subset being sent to the other parent. Finally, we return the resulting structures.

3.2 Mutation operators

3.2.1 Add Layer mutation

Function `add_layer_mut` is passed a genotype and an instance of the grammar. It chooses a random position inside the network, that is neither the first layer nor the last layer. It creates a new random

layer, using the grammar, and inserts this new layer in said position. This insertion causes the layer that was previously in that position to be moved one level to the right, in order to create space for the new layer. Function `enforce_in_out_feats` is invoked to ensure consistency between in/out features for adjacent layers. This ensures that the number of outputs from the previous level is consistent with the number of inputs for the new layer, and that the number of outputs from the new layer is consistent with the number of inputs for the layer that was pushed. The new network is then returned. Given that there was a requirement to not exceed 50 internal layers, we could check that we're not exceeding that limit. However, it was clarified that this requirement only applied to the initial instantiation, and that we should not prevent the networks from evolving to more than 50 internal layers.

3.2.2 Remove Layer mutation

Function `remove_layer_mut` is passed a genotype. The function tests if the network has internal layers that can be removed. If it only has two layers, those are mandatory and cannot be removed. In that case, the function returns the network unchanged. If there are more than 2 layers, the function randomly selects one internal layer (except the first and the last) and removes it from the network. Function `enforce_in_out_feats` is invoked to ensure consistency between in/out features for adjacent layers. The new network is returned.

3.2.3 Change Optimizer mutation

The Change Optimizer mutation consists in generating a new optimizer and replacing the existing one. Function `random_optimizer_generator` is passed the subset of the grammar that defines the optimizer. It is passed an extra parameter, establishing whether the return result should be in string or dictionary format. The function randomly chooses one of the defined optimizers and its set of parameters and returns this.

4 Experimental Setup

We followed the protocol that was given for the experiment:

1. We created five random networks;
2. We performed a cross-over of the first two networks, resulting in Crossover1 and Crossover2;
3. We performed Add Layer, Remove Layer, and Change Optimizer mutations on the remaining three networks.

The values for Accuracy and Loss over 50 epochs for each network are plotted in Figure 1.

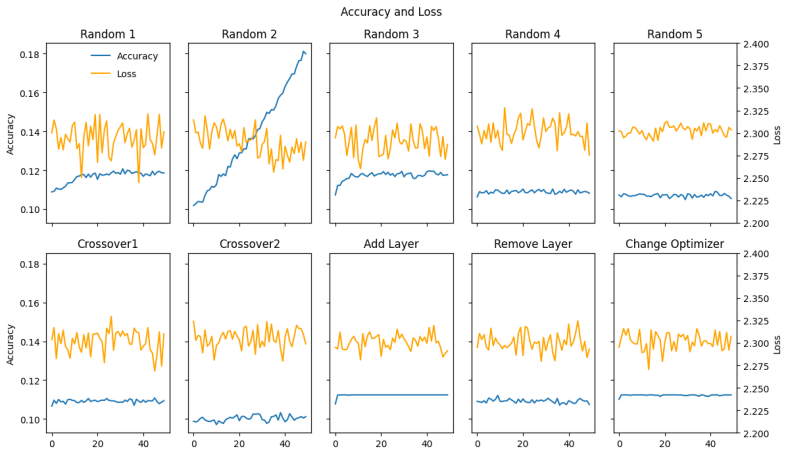

Figure 1: Experiment Results