Alex Santander nº 20220658, Diogo. Silva nº 20221393, Inês Rocha nº 20220052

Description of the classes and functions:

Generate_network_genotype function:

This code creates a random network string-based representation for the network, also called genotype, and returns the layers configurations, activation function, and optimizer details as a string representation of the genotype. The maximum number of layers used to create the neural network are 50, but only a random number of layers between 1 and 48 will be created, as the first and last layers are fixed linear layers with input or output size restrictions.

To create the representation, it is necessary to set a linear input layer size of 28*28 equivalent to a 2-dimensional image fitting the MNIST dataset. The output size and if it uses bias or not are randomly determined. The output layer has an output size of 10 for the MNIST dataset's final output (one for each number from 0 to 9).

The rest of the random number of layers are created with randomly chosen types of layers. Each options have their own parameter configuration. Also, the selection of activation functions and the optimizer with its parameters are chosen randomly.

Besides the parameter required on the script LayerNorm requires normalized_shape and BatchNorm1d require number of features, so they were added.

LayerNorm and BatchNorm1d have different normalization techniques, with LayerNorm normalizing independently for each sample, while BatchNorm1d normalizes along the batch dimension across all features.

The string representation of the neural network follows the format:

"Layer_type1|param1,param2;layer_type2|param1;(…):activation_function-optimizer|param1,param2"

The symbols are used as separators for the following processes, decoding the string and pass the parameters into the next function. Symbols:

- ';': Separates the different layers of the neural network.
- '|': Separates the layer and the optimizer from its parameters.
- ':': Separates the activation function from previous string(s).
- '-': Separates the optimizer choice from previous string(s).

This is an example of a genotype output the code generates:

'Linear|784,95,False;AlphaDropout|0.1527928779740244;LayerNorm|95,0.0006727208709194077;Linear|95,31,False; Linear|31,10,True:SELU-Adam|0.003957910099924216, 0.9148953345911256,0.9318540757079162'

Parameters used in this function:

For linear layers, three parameters are used: *N° of output features* (random.randint(10, 100)), *bias* (random.choice([True, False]), and Nº of input features, which always matches the Nº of output features of the previous lenear layer, exept when it is the first linear layer, in which case it will match the dataset. For BatchNorm1d, three parameters are used: *N° of output features* which match the number of features to the previous layer's output size, *eps* (random.uniform(1e-5, 1e-3)), and *momentum* (random.uniform(0.1, 0.9)). For LayerNorm, two parameters: *Normalized shape* which match the normalized shape to the previous layer's output size and *eps* (random.uniform(1e-5, 1e-3)). Finally, Dropout and AlphaDropout with one parameter: *Dropout probability(p)* (random.uniform(0.1, 0.5)).

The activation functions used are: [Sigmoid, ReLU , PReLU, ELU, SELU, GELU, CELU, SiLU]

Optimizer parameters used in this function						
Optimizer	Parameter			Optimizer	Parameter	
Adam	Lr	random.uniform(0.001, 0.01)		*NAdam*	Same as Adam	
	Beta1	random.uniform(0.9, 0.99)			Mo. decay	random.uniform(0.1, 0.99)
	Beta2			*SGD*	Lr	random.uniform(0.01, 0.1)
AdamW	Same as Adam				Momentum	random.uniform(0.1, 0.9)
	weight_decay	random.uniform(0.01, 0.1)			Nesterov	random.choice([True, False])

Adadelta	Lr	random.uniform(0.1, 1.0)				
	rho	random.uniform(0.8, 0.99)				

Net class:

This class takes the list of string layers provided by ***generate_network_genotype*** function, parsing its values and constructing neural network architecture based on the provided layer configurations.

The net *_innit*() was altered so that using *pase_layers*() all the strings are divided by ":" – meaning the layers – are parsed using *parse_layer_string*() and create layers with the parameters defined in the genotype; Then, in the *__innit_* function the *parse_activation_optimizer* function is used to parse the activation function and optimizer – this is done through the functions *parse_activation_function*(), and *parse_optimizer*() that create layers fitting the genotype instructions.

Training function:

Multiple neural networks are trained and evaluated in this process. Five networks with 50 epochs each were trained by generating random network genotypes and creating corresponding models. During each epoch, the models were trained using the training dataset in batches, and their parameters were updated based on calculated losses. The resulting losses and accuracies were recorded and saved for analysis. The code returns lists containing the recorded loss and accuracy values, which can be used to visualize the performance of the networks. The specific operators mentioned are used for creating new individuals but lack detailed explanation.

Change_optimizer_mutation function:

This function takes a network genotype as input and randomly mutates the optimizer type or its parameters. There is a 50% chance that the optimizer type will change. If the optimizer changes a new set of parameters is selected, and if the optimizer remains the same a new set of parameters is selected.

Remove_layer_mutation function:

This function removes a randomly selected non-linear layer from the network genotype. The code separates the layer and optimizer string and checks that the length of the total layers in the network is greater than 2. Otherwise, the code would delete either the input or output layer. A linear layer cannot be deleted as it would cause mismatches in the shapes of the network as a linear layer, BatchNorm1d, LayerNorm must match the output size of the previous linear layer – by deleting a linear layer they would no longer match.

***Add_layer_mutation function*:**

The Add Layer function adds a new layer to the network genotype. It separates the layer and optimizer string, checks if the model has less than 50 layers. The only layers allowed to be added are Dropout and AlphaDropout due to the dependencies mentioned in the *Remove_layer_mutation* function.

Crossover function:

This function performs crossover between two network genotypes by randomly selecting a subset from each genotype, finding the index range of each subset, swapping the subsets between the genotypes, and reconstructing the modified network genotypes. Only subsets that don't contain the fisrt layer, the last layer, or 'Linear', 'BatchNorm1d', and 'LayerNorm' layers are swapped - the subsets are found through the function find_consecutive_subsets.

Practical example:

The following example illustrate the process of creating 5 evolving neural network using genetic programming algorithms.

Then, operators will be applied to the 5 networks, 1 different type of mutation to 3 networks, and crossover to the 2 networks left.

This order of the operators will be:

- Network 1: Remove_layer_mutation function.
- Network 2: Add_layer_mutation function.
- Network 3: Change_optimizer_mutation function.
- Network 4 and 5: Crossover function.

Image 1: Loss and accuracy by model trained (no operators applied).

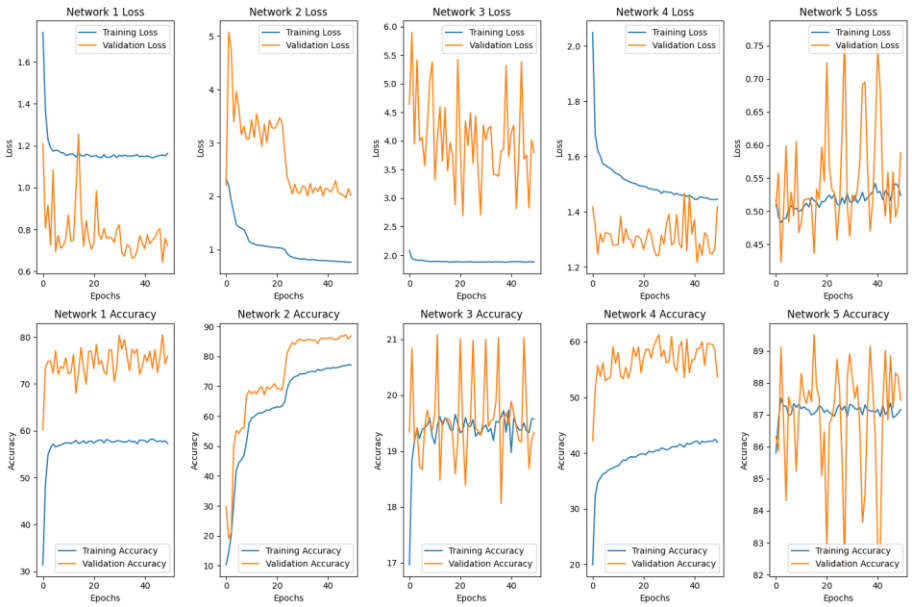
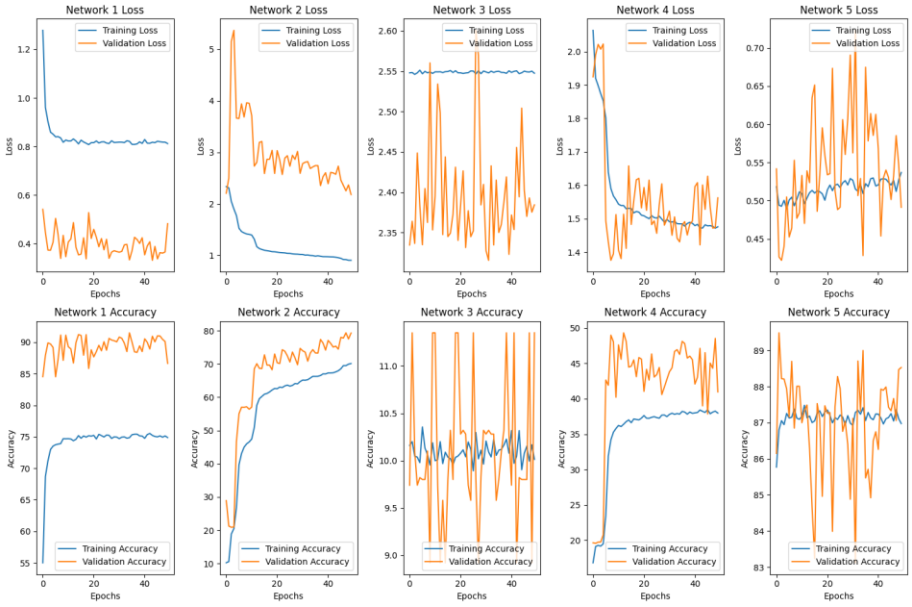


Image 2: Loss and accuracy by model trained (operators applied).



Loss function: Network 1 reduce the losses for train and validation. Networks 2 and 5 do not show significant changes. Network 3 improves on validation but is worst in train. Network 4 does not show significant improvement in train but is the worst in validation.

Accuracy: Network 1 improves in train and validation. Network 2, 3 and 4 decrease both in train and validation. Network 5 does not show significant changes.

Best model(s): Network 1 has a better overall result in loss and accuracy measures. Although, Network 2 has a good accuracy performance and the validation loss is higher than the train.

The results show that the code developed is capable of creating Neural Networks with good performance and that the operators have an impact on their performance. By using some sort of optimization or guidance, instead of taking the randomized approach, this could be a very powerful tool when trying to find the best structure of a Network.