Computational Intelligence ( )

Submission Assignment 5

Instructor: Jakub M. Tomczak *Name:* Martynas Vaznonis*, Netid:* mvs428

1 Introduction

Neural networks are a powerful set of algorithms which can learn complex nonlinear patterns in a wide range of different fields. But designing the architecture of a neural network is not an easy task as the problem is discrete and so non-differentiable. The problem of selecting appropriate neural network architecture is known neural, architecture search (Elsken et al., 2019). There are many methods developed for neural architecture search such as random search (Li and Talwalkar, 2020), reinforcement learning (Zoph and Le, 2016), and evolutionary algorithms (Liu et al., 2021).

In this paper, I examine the efficacy of evolutionary neural architecture search by implementing a simple evolutionary algorithm as a search strategy and compare the results with a benchmark architecture, developed manually beforehand.

2 Problem statement

For an evolutionary algorithm to work, it must be able to evaluate the individuals in a population and select the best ones based on one or more survivor selection methods (Dasgupta and Michalewicz, 1997). The best way to evaluate a neural network is to train it and then test it on the validation data. This, however, is computationally expensive and other methods have been developed for estimating the evaluation of a neural network (Xie et al., 2021), though not without their critique (Chu et al., 2021; Yu et al., 2019).

Fortunately, the data set used in this paper is the Optical Recognition of Handwritten Digits data set (Dua and Graff, 2017) which contains small instances, each containing only 64 features. This permits the use of the simplest and most accurate measure of the efficacy of a neural network which is to train it and test it on validation data.

Thus, the fitness function for the evolutionary algorithm is the evaluation of computed and trained neural networks. The objective of the neural networks generated by the evolutionary algorithm is to minimize the negative log likelihood loss. The objective of the evolutionary algorithm is to minimize the fitness of the neural network.

2.1 Potential issues

Despite the data set consisting of only small 8×8 pixel images with one color channel, the evolutionary algorithm has to evaluate hundreds of them which may still take a long time. This makes it difficult to select hyperparameters which is further complicated by the fact that hyperparameters need to be adjusted not only for the evolutionary algorithm but for the neural networks as well. Zela et al. (2018) show that hyperparamter and neural architecture searches may be combined to provide better results. This would also lead to a simplification in optimizing for hyperparameters since this would be done by the neural architecture search. Alas, this avenue is not explored in this paper.

3 Methodology

3.1 Genotype

The search space for this assignment is very limited as the neural network has to conform to a highly specific structure: Conv2d → f(.) → Pooling → Flatten → Linear 1 → f(.) → Linear 2 → Softmax.

The first layer must be a convolutional layer with 8, 16, or 32 filters and one of two structures.
Structure 1: kernel shape $= (3, 3)$, stride $= 1$, and padding $= 1$.
Structure 2: kernel shape $= (5, 5)$, stride $= 1$, and padding $= 2$.

So the encoding in the genotype for this layer contains two numbers, one binary for the structure and one integer, that takes the values between 0 and 2, which corresponds to the number of filters.

The second and sixth layers are activation layers, essential to introduce nonlinearity into the network. They can be ReLU, sigmoid, tanh, softplus, or ELU. To encode this, one integer value between 0 and 4 is enough to index which activation function to use.

The third layers is a pooling layer. It can either be max pooling or average pooling. It may also have a kernel size $= (1, 1)$ or kernel size $= (2, 2)$. Thus, two binary values are added to the genotype to represent the pooling layer.

The Flatten layer is invariant for all neural networks so it need not be added to the genotype.

The input of the first linear layers is wholly determined by the preceding layers so nothing needs to be encoded in the genotype for that. The output of the first linear layer is equivalent to the input of the second so only one value needs to be encoded for the both of them. Furthermore, the output dimensions of the network are always the same so nothing needs to be encoded about that either. The number of neurons in the second linear layer can be $10, 20, ..., 90, 100$. So the one integer encoding them can take the values in range $[1, 10]$.

The final layer is the softmax activation function, which much like the flatten layer, is invariant and so does not need to be encoded in the genotype.

The final genotype looks like $x = [c_f, c_s, f, p_t, p_k, n] \in [0, 2] \times [0, 1] \times [0, 4] \times [0, 1] \times [0, 1] \times [1, 10]$. Here, c_f and c_s are the number of filters and structure respectively in convolutional layer. Furthermore, f is the activation function and p_t and p_k are pooling type and pooling kernel size respectively. Lastly, n is the number of neurons in the second linear layer.

3.2 Phenotype

Given a genotype, a pytorch neural network module object can be generated or in other words the phenotype can be expressed. That object is trained and evaluated which provides the fitness value further used in the evolutionary algorithm.

3.3 Fitness function

There are two evaluation metrics which I consider for the role of fitness. These values are the validation loss and the validation classification error. This values are highly correlated so there may not be much of a difference between. Despite that, the reasoning for either appears, at least on the surface, logical. Classification error is the real value we want to minimize so it makes sense to use it as the fitness. On the other hand, the loss captures more detail than classification error. Small changes in loss may not manifest directly in classification error but be meaningful nonetheless.

There are other evaluation metrics which may be used such as accuracy which is just the inverse classification error, or the F1 value which captures more information than accuracy. These metrics would turn the evolutionary algorithm into a maximization algorithm and are not examined in this paper.

3.4 Benchmark architecture

The architecture of the benchmark network was created with the following structure:

```
0: Conv2d(1, 16, kerel size=(3, 3), stride=1, padding=1)
1: MaxPool2d(kernel size=2, stride=2)
2: Conv2d(16, 64, kernel size=(3, 3), stride=1, padding=1)
3: MaxPool2d(kernel size=2, stride=2)
4: Flatten()
5: Linear(in feature=256, out features=128)
6: ReLU()
7: Linear(in features=128, out features=10)
8: Softmax(dim=1)
```

It does not conform to the general structure outlined in 3.1. This larger structure is used to make it more difficult for the evolutionary algorithm to find a network that can beat it. This way, if the performance of the model generated by the evolutionary algorithm surpasses this benchmark, it will be a substantial victory as it would show that even a highly constrained model, when optimized, can surpass unconstrained human-designed models.

3.5 Evolutionary algorithm

3.5.1 Parent selection

There are a number of parent selection mechanisms. The simplest is to pick the parents at random as they already are the selected and most appropriate individuals of their generation. Alternatively, the parents may be selected on fitness and ranking. There is some debate about whether that actually has a positive effect over random selection (Spears et al., 1993).

As it is unclear if a specialised parent selection would improve the convergence to an optimal solution, a simple random parent selection is implemented.

3.5.2 Recombination

The crossover operator recombines different parents to form the children of the next generation. The way it is implemented in this experiment is:

```
for i in N:
    select random parent $x_j$
    generate integer $k \in [0;6)$
    $x_{i,new}$ = concatenate($x_{i\%P}[:k], x_j[k:]$)
```

Here, N is the size of the population and P is the number of parents. The size of the contribution from either parent is determined by k. If k is 0, then new x_i is equivalent to x_j and if $k = 5$, new $x_i = x_{i\%P}$. For other values of k, $x_{i,new}$ has attributes of both parents. This is an instance of one-point crossover.

3.5.3 Mutation

The mutation operator is an essential part of the evolutionary algorithm used to introduce new values in the population. For every data type used in encoding the genotype there is a number of different mutation operators devised (Koenig, 2002).

Since the genotype used in this experiment is constituted of multiple different data types, multiple mutation operators must be combined.

- For the binary parts c_s, p_t, and p_k, the mutation operator is flipping bits. First, sample mask $m \sim \texttt{Bernoulli}(m|\Theta)$ and then perturb $x_{new} = (x + m)\%2$.
- For the integer values c_f and n, creep mutation is used. First, a discrete approximation of a Gaussian is sampled, $m \approx \mathcal{N}(0, \sigma^2)$, then $x_{new} = x + m$.
- For a categorical encoding using an integer, creep mutation does not make sense as there is no distance between the values. The value 0 is not further away from the value 4 as it is from the value 1, they simply encode different categories. For this reason, the mutation operator for f is a simple random integer generator.

3.5.4 Survivor selection

For survivor selection, a simple elitist strategy was chosen. The children and parents are combined and the best N individuals are selected as the new generation. Although elitism scales poorly (Blickle and Thiele, 1996), for a small, discrete search space it suffices and achieves appropriate results.

4 Experiments

4.1 Neural network hyperparameters

Some hyperparameters had only negligible changes in the efficacy of neural network. These are batch size and learning rate which were then kept identical to the benchmark network at 64 and 10^{-3} respectively.

The weight decay hyperparameter was set to 10^{-4} which is an order of magnitude higher than what it was for the benchmark network. This was done to substantially increase the loss with l2 regularization and reduce overfitting.

Finally, the number of epochs for training was the toughest hyperparameter to tune because it had to provide a somewhat consistent evaluation of the model but not take overly long so that the evolutionary algorithm could terminate. Furthermore, the larger the number of epochs, the likelier overfitting becomes. Ultimately, 50 epochs were chosen as the optimal number to use for training. Additionally each genotype was used to generate five identical models, which were trained and evaluated, and then the average of the evaluations was used for the fitness. This was done to further improve the consistency of the fitness for a specific genotype.

4.2 Evolutionary algorithm hyperparameters

The population size cannot be too large as every individual in the population has to be trained and evaluated 5 times for 50 epochs each. Thus, for every additional individual in the population, each generation has to complete an additional 255 epochs. A good number was found to be 20 individuals.

The number of generations succumbs to a similar problem where every generation adds substantial time required to run the algorithm. Multiple runs have indicated that 10 generations usually suffice to converge to a good solution. The final number of generations was chosen to be 20 to help insure complete convergence.

The standard deviation used for the creep mutation was left as 1. With $\sigma = 1$, most mutations are changes of magnitude 0 or 1 with occasional leaps of magnitude 2 or 3. These, values are appropriate for small ranges like those of c_f and n.

The hyperparameter number of parents P was set equal to N so that all parents participate in creating the next generation.

4.3 Running the algorithm

The initial population was generated at random within the bounds specified in 3.1 and subsequently evaluated. The evolutionary algorithm works with the following structure:

```
for i in number of generations:
    $x_{parents}, f_{parents}$ = parent_selection($x, f$)
    $x_{children}$ = recombination($x_{parents}, f_{parents}$)
    $x_{children}$ = mutation($x_{children}$)
    $f_{children}$ = evaluate($x_{children}$)
    $x, f$ = survivor_selection($x, x_{children}, f, f_{children}$)
```

With each generation, the individuals converge to a good, potentially optimal, solution. The best structure found by the algorithm should rival the hand made benchmark.

After running the algorithm with loss as the fitness function and with classification error as the fitness function I found that the difference in quality of the result is negligible. Because of this, the classification error was chosen arbitrarily as the fitness function.

5 Results and discussion

5.1 Generations

The convergence towards a good solution can be seen in figure 1. Though the the improvement looks marginal only dropping ~ 0.01 from ~ 0.033 to ~ 0.023, the results are very positive. The reason for such a small drop is because the basic structure discussed in 3.1 is already a good model. The optimization can then only provide small improvements which can clearly be seen.

The scatter plots in figure 2 show where the points of different generations land. Some noise has been added to each point so overlapping points can be seen. The points of most interest are the red ones as they are the final generation.

The first plot shows the position of c_f and n. Almost every red point has $c_f = 2$ which means that the number of filters prioritized by the evolutionary algorithm is 32 over 16 and 8. This makes intuitive sense as the larger the number of kernels, the more features the convolutional layer will be able to extract. The number of neurons in the second linear layer is always more than 50 but besides that there is no clear winner as there are many individuals in each category.

The second plot shows the position of c_f and f. There appear to be 3 clear winners for the activation function. It is the ReLU, the tanh, and the ELU. It is surprising to see such a large gap between sigmoid and tanh considering how similar they are. The best of those 3 appears to be the ELU function.

The third plot shows the position of c_s and c_f. Almost every red point has $c_s = 1$ which corresponds to the structure where kernel shape $= (3, 3)$, stride $= 1$, and padding $= 1$. The result shows that a smaller kernel works better in this instance. This may be because the digits are so small (8×8) or a smaller kernel may be generally better.

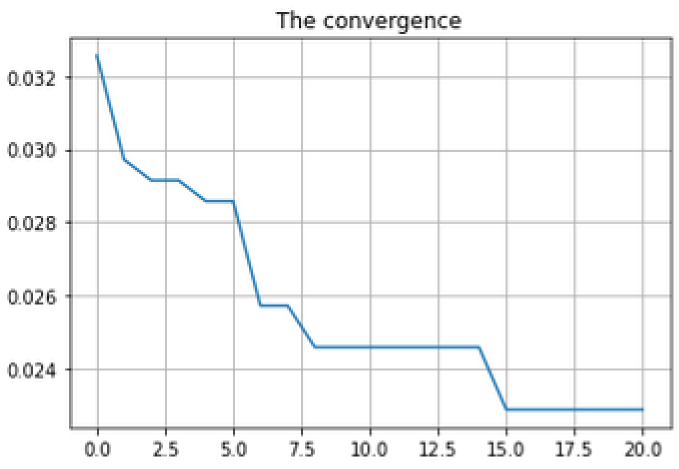

Figure 1: Convergence of the evolutionary algorithm over 20 generations

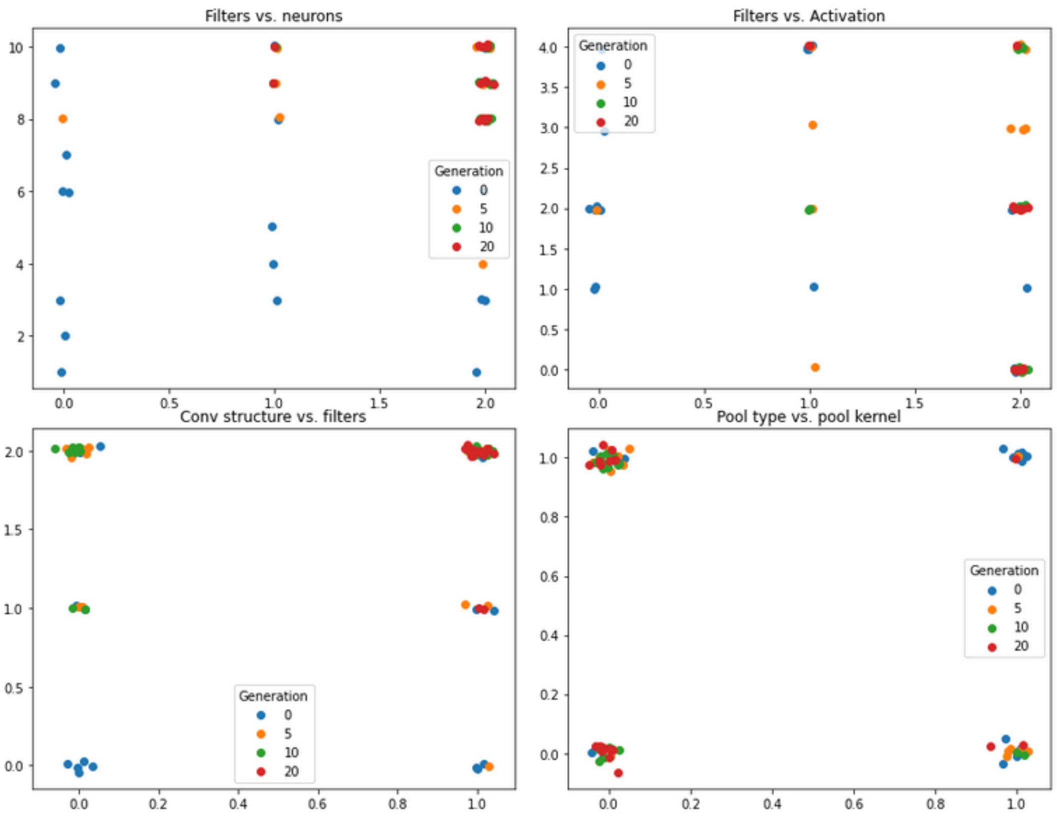

Figure 2: Plots used to display the positions of different generations

The final plot shows the position of p_t and p_k. The vast majority of red points have $p_t = 0$ which corresponds to the max pooling type. It is surprising to see that max pooling is so much better than average pooling considering that it only perpetuates a fraction of the information contained within the kernel while average pooling is influenced by the whole kernel. The reason for this may be that max pooling only keeps the most salient and important values and everything else is considered noise. Furthermore, the red points are evenly split between $p_k = 0$ and $p_k = 1$. This shows that despite substantially reducing dimensionality, the pooling layers do not give up valuable information.

5.2 Model

The very best model that evolutionary algorithm came up with after 20 generations took the following structure:

```
0:  Conv2d(1, 32, kernel size=(3, 3), stride=1, padding=1)
1:  ELU()
2:  MaxPool2d(kernel size=2, stride=2)
3:  Flatten()
4:  Linear(in features=512, out features=70)
5:  ELU()
6:  Linear(in features=70, out features=10)
7:  Softmax(dim=1)
```

This is a substantially smaller model than the benchmark. Despite that, it still manages to marginally outperform the benchmark. Figure 3 shows the training and performance of the generated model. The test classification error is 0.0537. Figure 4 shows the training and performance of the benchmark model. The classification error for it is 0.06. The test loss is smaller on the benchmark but that can be explained by the generated model having 10 times the penalty for the magnitude of the weights.

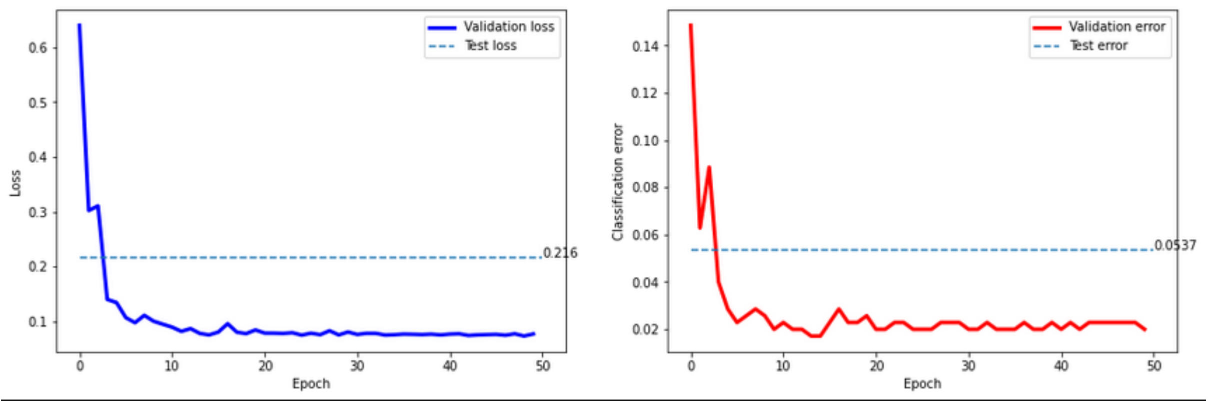

Figure 3: Performance of the evolutionary algorithm generated model

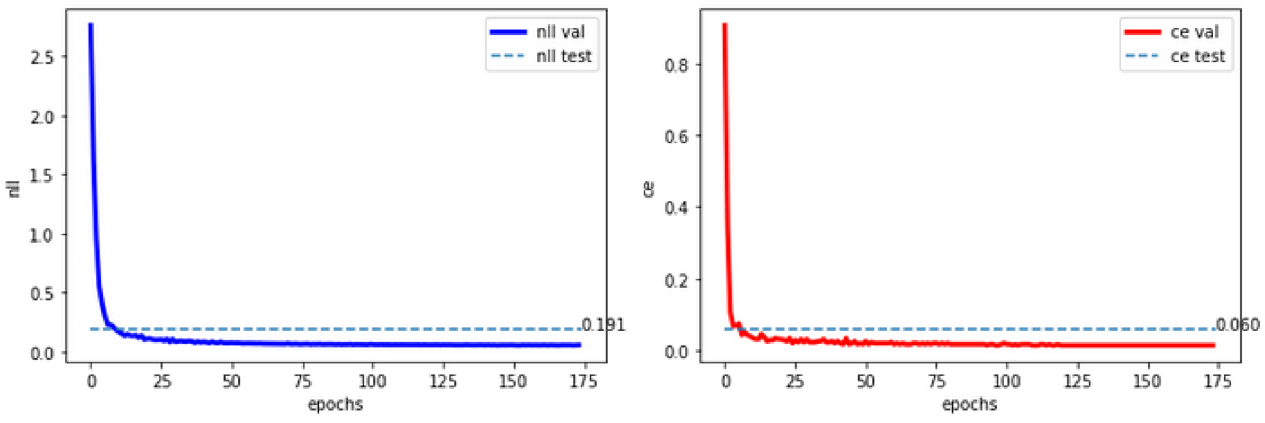

Figure 4: Performance of the benchmark model

5.3 Conclusions

This paper examined evolutionary neural architecture search on a very limited search space. Even so, the result was a better network in both performance and speed than a handcrafted network with greater flexibility as it was not constrained by a set number of layers.

This particular evolutionary algorithm could not be scaled as it stands to generate more complicated networks and so the conclusions are limited. An unconstrained search space could lead to more interesting discoveries such as dimensionality reduction without the use of pooling versus with pooling. Having the algorithm try a variable number of different layers with different parameters in any possible configuration would take a very long time to run but the results could lead to new discoveries in optimal neural network architecture design.

References

Blickle, T. and Thiele, L. (1996). A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation*, 4(4):361–394.

Chu, X., Zhang, B., and Xu, R. (2021). Fairnas: Rethinking evaluation fairness of weight sharing neural architecture search. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 12239–12248.

Dasgupta, D. and Michalewicz, Z. (1997). Evolutionary algorithms—an overview. *Evolutionary Algorithms in Engineering Applications*, pages 3–28.

Dua, D. and Graff, C. (2017). UCI machine learning repository.

Elsken, T., Metzen, J. H., and Hutter, F. (2019). Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017.

Koenig, A. C. (2002). A study of mutation methods for evolutionary algorithms. *University of Missouri-Rolla.*

Li, L. and Talwalkar, A. (2020). Random search and reproducibility for neural architecture search. In *Uncertainty in artificial intelligence*, pages 367–377. PMLR.

Liu, Y., Sun, Y., Xue, B., Zhang, M., Yen, G. G., and Tan, K. C. (2021). A survey on evolutionary neural architecture search. *IEEE transactions on neural networks and learning systems.*

Spears, W. M., Jong, K. A. D., Bäck, T., Fogel, D. B., and Garis, H. d. (1993). An overview of evolutionary computation. In *European conference on machine learning*, pages 442–459. Springer.

Xie, L., Chen, X., Bi, K., Wei, L., Xu, Y., Wang, L., Chen, Z., Xiao, A., Chang, J., Zhang, X., et al. (2021). Weight-sharing neural architecture search: A battle to shrink the optimization gap. *ACM Computing Surveys (CSUR)*, 54(9):1–37.

Yu, K., Sciuto, C., Jaggi, M., Musat, C., and Salzmann, M. (2019). Evaluating the search phase of neural architecture search. *arXiv preprint arXiv:1902.08142.*

Zela, A., Klein, A., Falkner, S., and Hutter, F. (2018). Towards automated deep learning: Efficient joint neural architecture and hyperparameter search. *arXiv preprint arXiv:1807.06906.*

Zoph, B. and Le, Q. V. (2016). Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578.*