Research Day, April 22, 2022

Using Evolutionary Algorithms to Optimize Hyperparameters for Keras Deep Learning Models to Solve the Two Intertwined Spiral Problem

Mark Kocherovsky, MS Computer Science Candidate; CJ Chung, PhD, Professor
Department of Math & Computer Science, College of Arts and Sciences

ABSTRACT

Deep learning models have a set of associated hyperparameters that can be thought of as tuning knobs. As the hyperparameters specifically determine a neural network's structure, they cannot be learned from training data. The programmer does not know the best model structure to use for each problem out of hand. They can use rules of thumb, references on similar examples, or manual trial-and-error to find the best model. Here we propose the use of Evolutionary Strategies to find optimal hyperparameters for densely-connected neural networks to solve the Two Intertwined Spiral Problem as an example. In this study, we optimize the number of hidden layers, the number of neurons in each hidden layer, the learning rate, the choice of activation function, the batch size, the choice of the loss function, the choice of the optimizer algorithm, and the number of training epochs. The results show that this approach is an effective way to design and find optimized neural network models.

INTRODUCTION

An artificial neural network (ANN) is a data structure designed to perform machine learning algorithms. A network is “trained” on a set of example data and target outcomes in order to mimic patterns between the input and intended output [1]. However, the programmer does not know which ANN hyperparameters (structural elements) are optimal for any given problem. They can look at reference materials, but if a problem is entirely new, their best course of action might be trial and error. This can be a very time-intensive process.

A solution to this problem comes in the form of **evolutionary strategies** (ES). ES algorithms, more specifically ES(1+1), works by starting with a parent value or set of values. This can be any value for any function that the programmer wants to optimize. The parent is evaluated on the intended function to find its **fitness**. A child is then produced by copying and mutating (tweaking) the parent values. If the child has a better fitness than the parent, then the parent is discarded and replaced by the child. If the child performs worse than the parent, the child is discarded and the parent is re-used. This continues for some number of **generations** until either an acceptable (guaranteed to be near-optimal) solution is found or the program has exceeded the maximum number of generations. Multiple trials of the algorithm can be run until a solution is found or the program has run the set number of trials. This is advisable due to the random nature of ES programs. The **⅕ success rule** can also be applied to the problem to better fine-tune searching. The rule states that if the program has a certain ratio of successes (better performing children) to attempts (defined by a certain **window size**) or higher, then the **step size** (range of mutation) should be increased to expand the search space, perhaps finding more solutions. If the ratio is worse, then decrease the step size and find a more nuanced approach. It is called the ⅕ rule because the ratio should be successes to one-fifth of the window size [2].

We show that ES(1+1) can be used to find near-optimal parameters for neural networks that can solve the **two-arm intertwined spiral problem**. The non-linear nature of the problem can make it difficult to find the optimal hyperparameters, but we have achieved this goal using ES(1+1) 1/5.

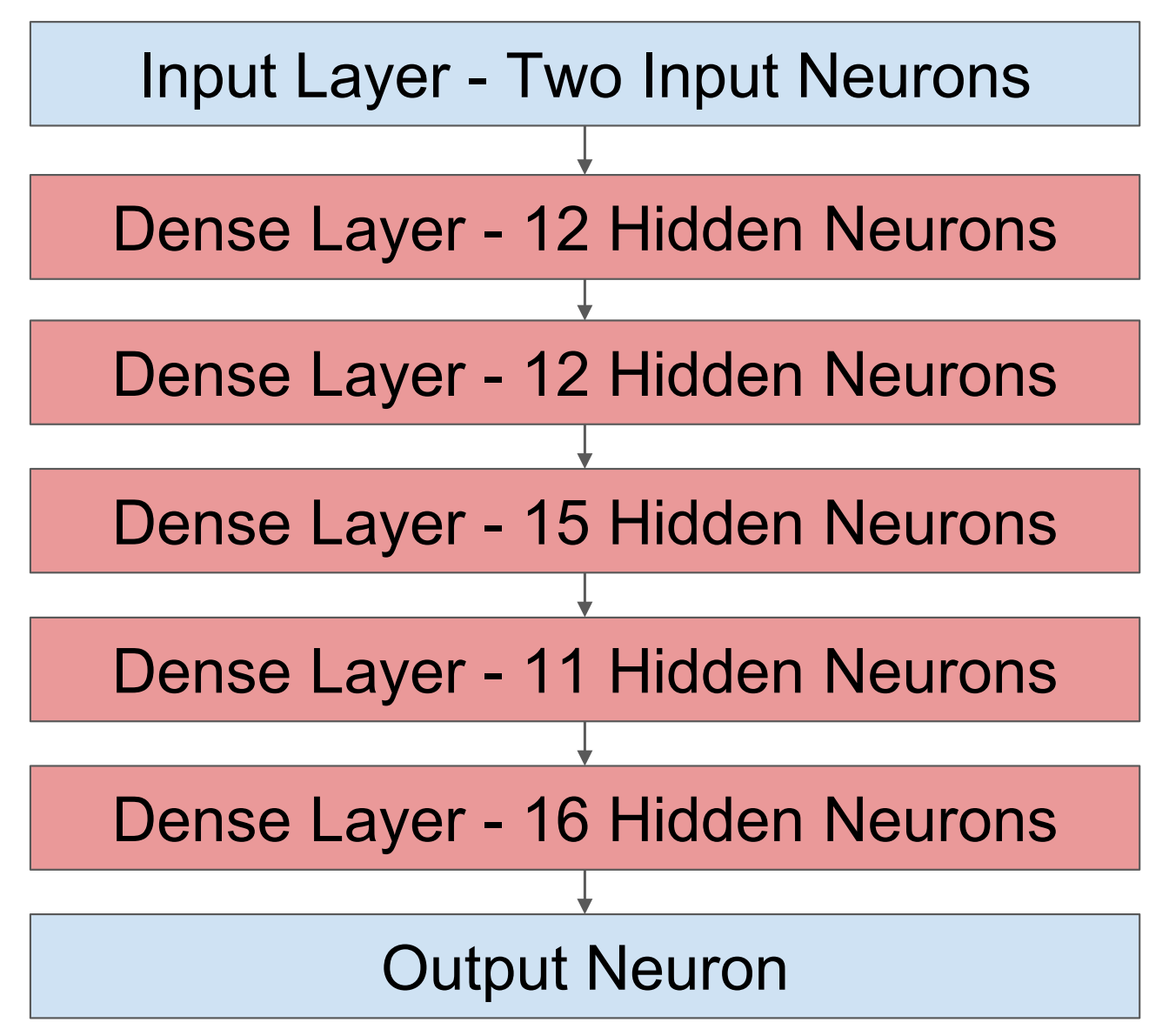
Input Layer - Two Input Neurons
Dense Layer - 12 Hidden Neurons
Dense Layer - 12 Hidden Neurons
Dense Layer - 15 Hidden Neurons
Dense Layer - 11 Hidden Neurons
Dense Layer - 16 Hidden Neurons
Output Neuron

Figure 4. The structure of the final model after being trained on 380 epochs. In addition to these parameters, the model also uses a learning rate of 0.01, the exponential linear unit as the activation function, a batch size of 4, mean squared error as the loss metric, and the RMSProp optimizer.

DESIGN

We use the Tensorflow and Keras libraries in Python 3 to evolve a set of hyperparameters for neural network design [3,4]. We mutate the following hyperparameters: number of hidden layers in the network (1-7), number of neurons in each hidden layer separately (1-64) (together these parameters control the amount of logical processing units in the network); the learning rate, which determines the rate at which parameters (network weights) are tweaked between predictions during training; the activation function, which determines how output values are “squashed” before moving to the next layer; the batch size, which determines how many samples are passed to the network at once during training; the optimization algorithm used to minimize error; and the number of epochs the network is trained on.

There are up to 15 trials, each of which has up to 35 generations. The program would stop once an acceptable solution (which gets an error within 0.02) is found. The model is trained on a set of 400 points where each class has 200 data points. The program generates a new dataset each time it is ran. The training dataset is shown in Figure 1.

The x-coordinate of each point is calculated using the formula 0.1mt*cos(t)+x_i, and the y-coordinate of each point is calculated using the formula 0.1mt*sin(t)+y_i, where m is a value controlling the direction (1.0 for red and -1.0 for blue), t is a random sample from the normal distribution (valued between 0 and 10), and x_i and y_i are sampled from a gaussian distribution with a mean of 0.0 and a variance of 0.03. The model takes in the x and y coordinates as input, and outputs the expected class value (between 0 for red and 1 for blue).

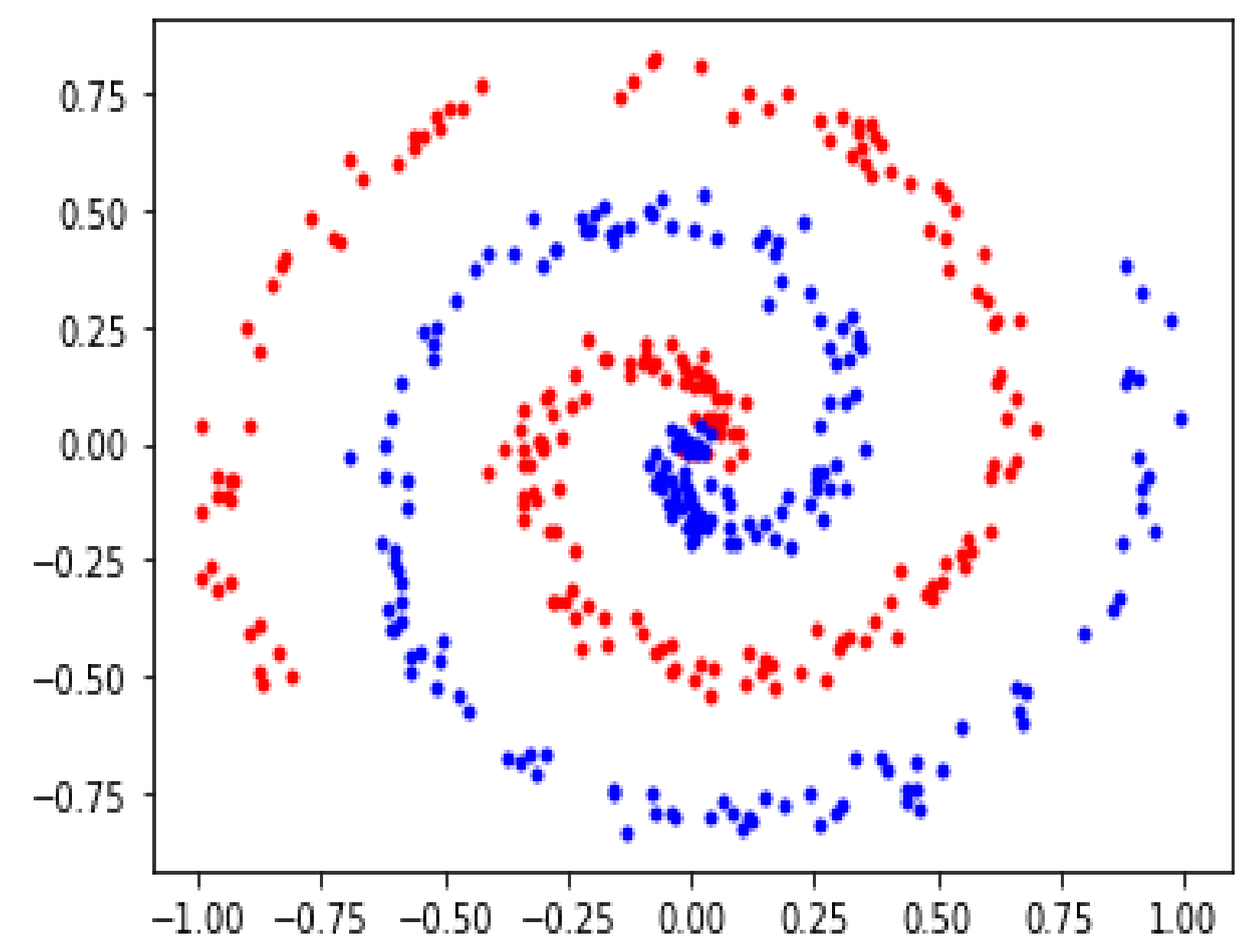

Figure 1. Training dataset for the final model.

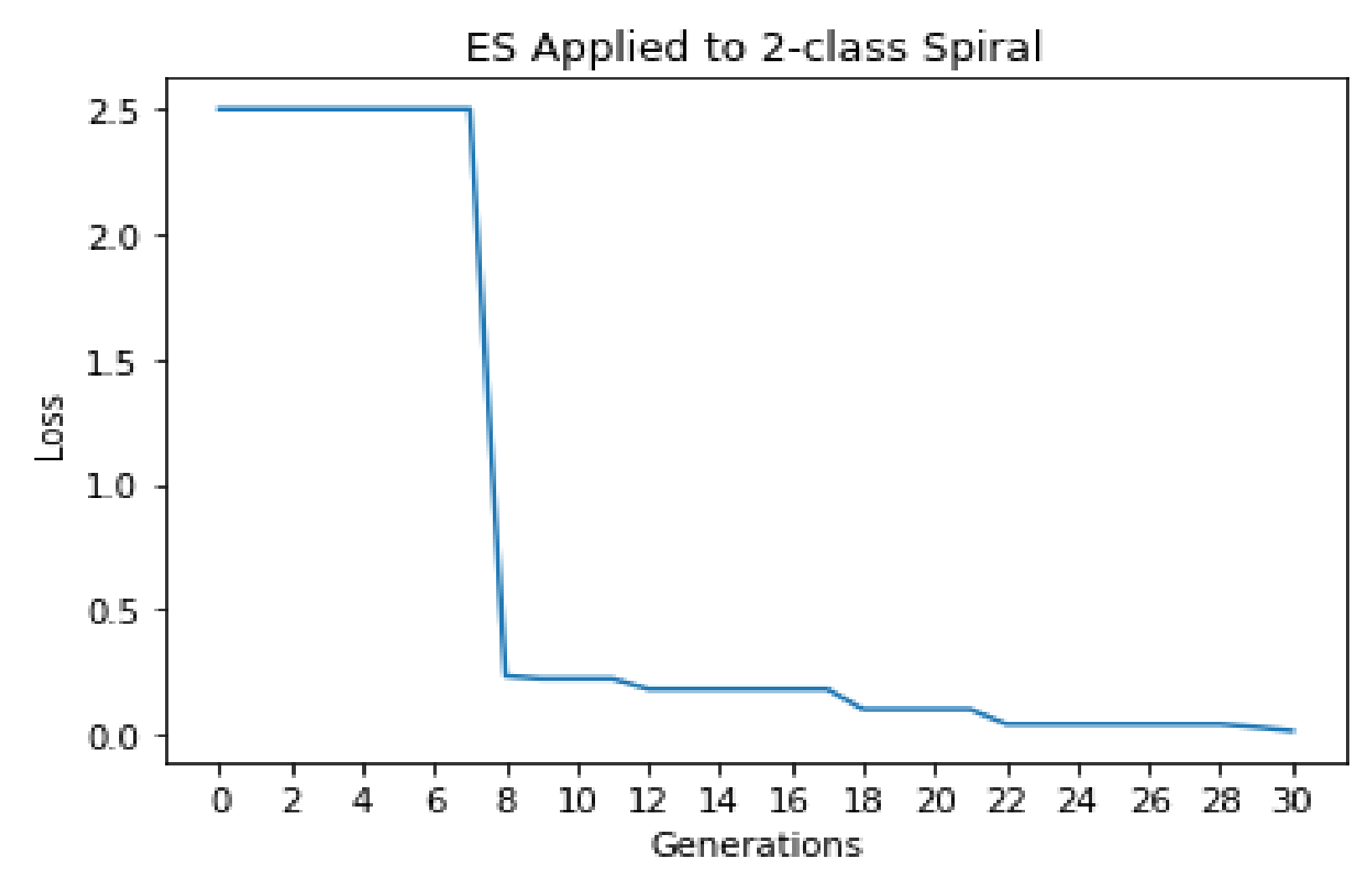

Figure 2. Loss for each generation for the final model.

After a near-optimal model is found, it is tested on a randomly generated test set with 500 points in each class (1000 total), and its accuracy is measured. Finally, the model made predictions on a grid of points from $-1.0 \le x \le 1.0$ and $-0.75 \le y \le 0.75$ to see the overall shape of the model's training.

The final model was trained using 380 epochs and had an average mean squared error of 0.01400. The loss over time for the model's evolution is shown in Figure 2.

RESULTS

The model was able to make predictions on the test set with **98.10% accuracy!** This indicates a very well trained network. A diagram of the test set next to the model's predictions is shown in Figure 3. Table 1 shows the hyperparameters for the best model and the range across seven accepted tests. An accepted test produces a sufficiently optimal model. Figure 4 shows a diagram of the best model. Figure 5 shows the model's results when applied to a grid of points. A spiral shape is clearly visible.

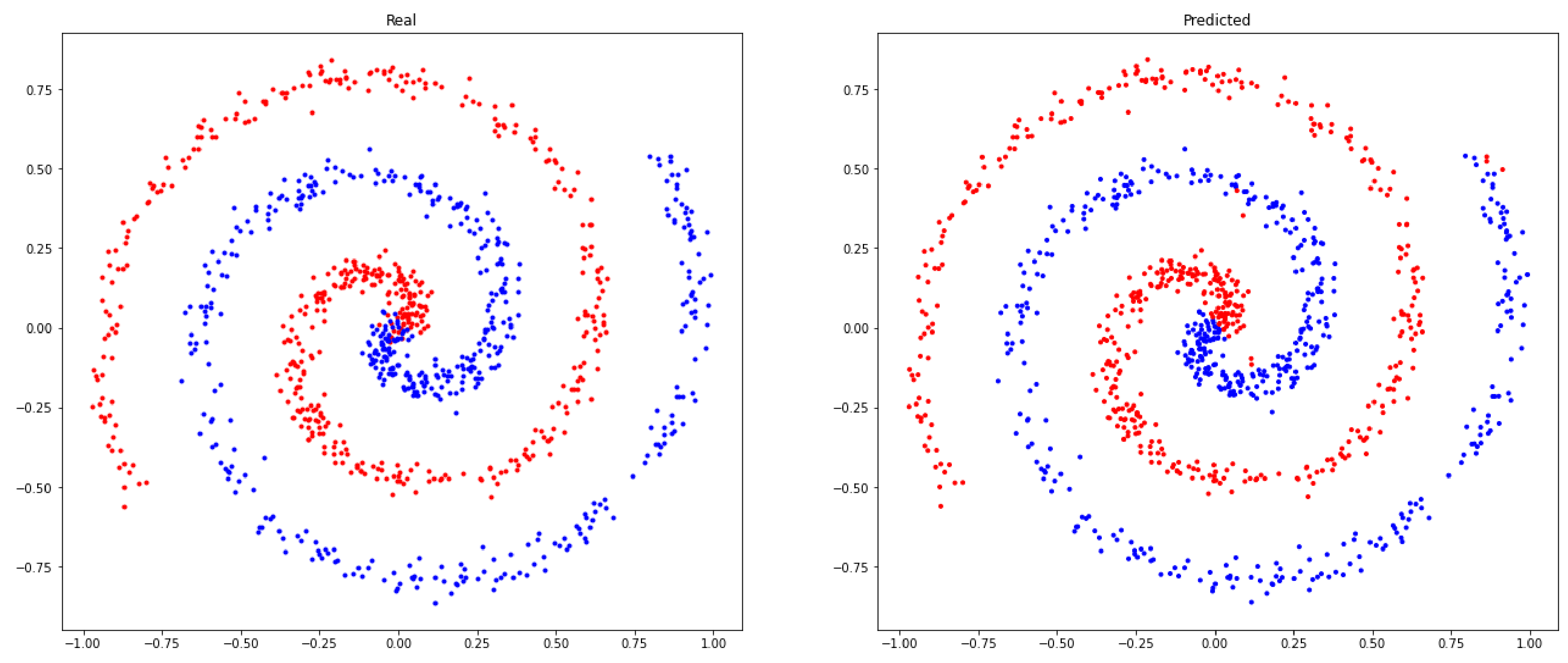

Figure 3. Left: the randomly-generated testing set that the model is tested on. **Right:** the predictions returned by the model for each point in the testing set. It is mostly accurate, but does make some errors. If you cannot easily tell the difference between the two pictures, then the model has done its job!

Hyperparameter	Best Model Value	Range Across Tests
Generations	30	[2, 34]
Error	0.01400	[0.00568, 0.01755]
# of Hidden Layers	5	[2, 5]
# of Neurons per Layer	12 | 12 | 15 | 11 | 16	[11, 15]
Learning Rate	0.01	[0.01, 0.05]
Activation Function	Exponential Linear Unit	See Figure 6, G1
Batch Size	4	[1-4]
Loss Function	Mean Squared Error	See Figure 6, G2
Optimizer	RMSProp	See Figure 6, G3
Epochs	380	[380, 400]

Table 1: Table of the final network's hyperparameters and the range of hyperparameter values seen across **seven accepted** (produced a sufficiently optimal network) **trials**. A diagram of the structure can be seen in Figure 4. For the three discrete values; the activation function, loss function, and optimizer, see the three frequency graphs on the poster.

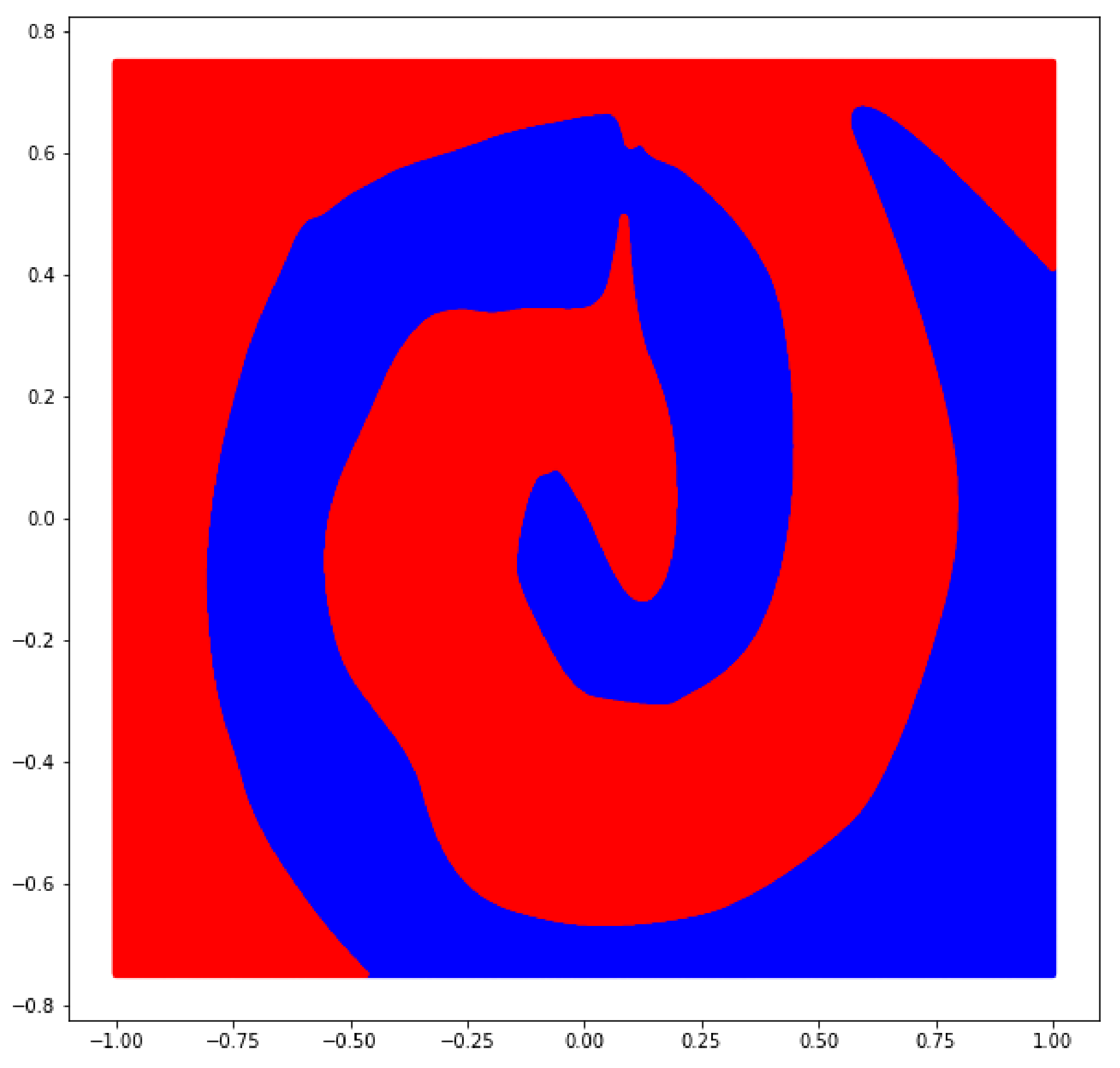

Figure 5. Diagram of the model's prediction on each point on the given grid. This shows the overall shape of the spiral it has learned. The edges of the spiral, particularly on the top right, are mainly a guess based on adjacent data. Other than that and the red pincer in the top-center, it approximates a spiral. It can also be seen that the shape roughly approximates the training data, and any gaps in the training data explain gaps in the shape.

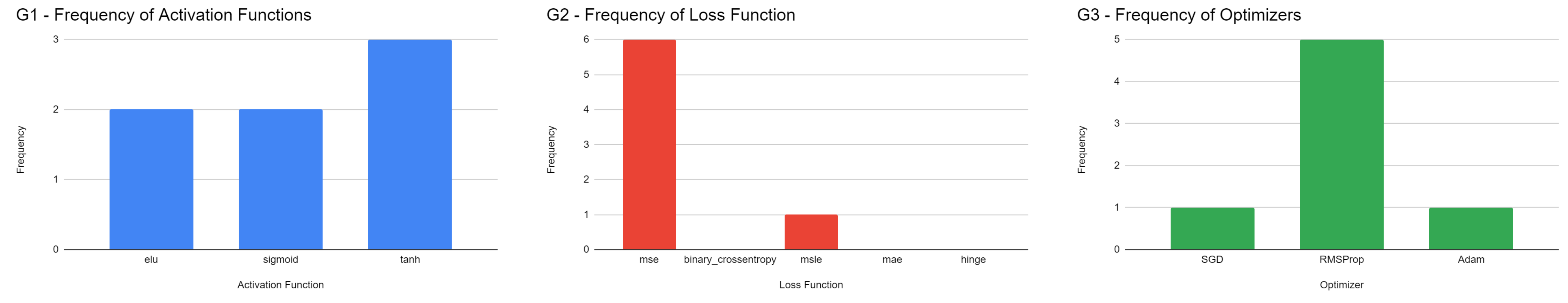

Figure 6. G1: Frequency of activation functions across seven accepted tests. It seems that all three of them are viable strategies. G2: Frequency of loss functions across seven accepted tests. Mean squared error (MSE) clearly dominates. G3: Frequency of optimizers across seven accepted tests. RMSProp clearly dominates, though SGD and Adam have been successfully used in one case each.

DISCUSSION

Our results demonstrate that ES(1+1) with ⅕ success rule can be used to solve the intertwined spiral problem by finding a near-optimal neural network to distinguish between two arms of the spiral. The automation of iterative neural network design can be used to improve productivity for researchers and AI engineers. Though this can still take a long time, the guided random optimization can be far more precise than human trial-and-error, thus improving productivity. As computing power continues to improve, ES-developed neural networks could very well become the norm for neural network development.

REFERENCES

[1] F. Chollet. *Deep Learning with Python*, 1st ed. Apress: Berkeley, CA, 2017; Manning
[2] M. Schumer, K. Steiglitz. Adaptive Step Size Random Search, in IEEE Transactions on Automatic Control, Vol. AC-13, No. 3, (1968). found in https://arxiv.org/pdf/2006.11026.pdf
[3] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah,M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Tal-war, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Víegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, TensorFlow: Large-scale machine learning on heterogeneous systems (2015), software available from tensorflow.org.
[4] M. Gorsline, J. Smith, C. Merkel. On the Adversarial Robustness of Quantized Neural Networks. (2021).