Complex Valued Deep Neural Network Module

Vladimir Lunić[1]

[1]Department of Computation and Automation, University of Novi Sad
Trg Dositeja Obradovića 6, Novi Sad, Serbia
[1]`vladimir.lunic2000@gmail.com`

***Abstract*— The paper details the implementation and evaluation of a deep neural network that uses only complex values for its parameters and input, hidden and output layer neurons. The paper also provides multiple activation functions in the complex domain and a recommendation for which performed best, likewise for the loss function. On top of this, the paper introduces a novel method for classification in the complex domain that relies solely on the properties of the complex domain. The network was evaluated on multiple synthetic datasets showing results on par with a real valued deep neural network, but with two times less parameters for those tasks. These tasks included a binary classification problem and a multiclass classification problem.**

***Keywords*— Artificial Intelligence, Machine Learning, Deep Learning, Neural Networks**

I. Introduction

Current methods of handling complex valued data in the world of machine learning is to unravel each complex number either into its real and complex parts or into its magnitude and angle, thus inherently introducing double the amount parameters a model will have to learn, as well as doubling the number of neurons required per layer in deep learning. With the current state of the art models growing larger, same with input vectors' sizes, there is no space for doubling the size of everything. This paper proposes a solution for complex valued datasets that utilizes the fact that the input is in the complex domain,to its advantage. The proposed solution uses complex valued neurons in all layers of the network, complex valued weights and biases, as well as turning any other parameters that are present in the network into a complex value. Other than those, it uses activation functions tailored to the complex domain, and loss functions from the real domain, modified to work in the complex domain.

This paper will outline the mathematical reasoning behind why the model works much the same way as its real counterpart, detail the implementation of the module that will be provided as an annex to this paper, and evaluate the model on synthetic dataset tasks created for comparison with a real valued neural network.

TABLE I

Symbols and Notations

i	Imaginary unit
z	Complex number
$\mathbb{R}$	Set of real numbers
$\mathbb{C}$	Set of complex numbers
$\mathbb{X}$	Arbitrary set of numbers
Re	Real component
Im	Imaginary component
r	Magnitude
φ, θ	Angle
e	Euler's constant
η	Learn rate
∇	Gradient
x	Arbitrary variable
$\mathcal{L}$	Loss function
$\hat{y}$	Neural network output
y	Label

II. Complex Domain Operations

This section aims to show basic operations that will be used throughout the complex valued neural network model.

A complex number can be represented in multiple ways. The ones that will be used most throughout this paper will be the component representation, $Re(z) + iIm(z)$, and the Euler representation, $re^{i\varphi}$. From which the component representation will be most used throughout as that is the way that complex numbers are represented in data most often. It is also the easiest for the processor to operate on since it involves the least complex processor operations as opposed to having to constantly use exponentiation and multiplication with the euler representation.

A. Addition

Addition in the complex domain is computed by adding the real and imaginary parts of both constituents separately.

$$z_1 + z_2 = Re(z_1) + Re(z_2) + i(Im(z_1) + Im(z_2))$$

B. Subtraction

Subtraction is done much the same way as addition, but instead of adding the components separately, they are subtracted.

C. Multiplication

Multiplication is performed by distributing the component representation of the two constituent complex numbers

$$Re(z_1z_2) = Re(z_1)Re(z_2) - Im(z_1)Im(z_2)$$
$$Im(z_1z_2) = Re(z_1)Im(z_2) + Re(z_2)Im(z_1)$$

It needs to be noted that when using multiplication both the real and complex components of the complex numbers being multiplied are used to compute the product's real and complex components, as opposed to addition and subtraction wherein for calculating the new real component only the real components of numbers being added are used, same for the imaginary component. This means that the imaginary components influence the resulting real component and vice versa.

D. Division

Division in the complex domain requires the imaginary unit not to be present in the denominator thus resulting in the expansion of the denominator by its complex conjugate to remove the imaginary unit resulting in the real and imaginary components being calculated as the following formulae show.

$$Re\left(\frac{z_1}{z_2}\right) = \frac{Re(z_1)Re(z_2)+Im(z_1)Im(z_2)}{Re(z_2)^2+Im(z_2)^2}$$
$$Im\left(\frac{z_1}{z_2}\right) = \frac{Re(z_1)Im(z_2)-Re(z_2)Im(z_1)}{Re(z_2)^2+Im(z_2)^2}$$

Similarly to multiplication, in division, imaginary components of the denominator and numerator influence the resulting real component, and vice versa for the imaginary component of the result.

E. Scalar Operations

Addition and subtraction of a real valued scalar from a complex number will result in only the real component being changed in the result of the operation, which is directly derived from the formula for complex addition and subtraction, as a real valued scalar has the imaginary component equal to zero.

This does not apply to multiplication and division. When multiplying a complex number by a real valued scalar, both the real and imaginary components are scaled, and with division the real component and the imaginary component are divided by the scalar separately.

This is an important fact to keep in mind when applying multiplication and division on complex vectors and numbers, as it may have different effects whether a complex vector is multiplied by a real valued scalar or a complex number.

F. Linear Algebra in $\mathbb{C}$

For the purposes of this paper, all the linear algebra used in machine learning and neural networks from $\mathbb{R}$ is the same in $\mathbb{C}$. The only difference being that complex operations shown in this section are used on vectors that have complex valued components, but multiplication and division of vectors and matrices has to be kept in mind.

III. Complex Valued Activation Functions

There are multiple types of proposed activation functions seen in multiple papers [2-13]. Most can be categorized into Component split type activations, parameter split activations and fully complex activations. Component split type activation functions split the complex number into its Re and Im components then apply an activation function that is normally used on a real valued neural network and then combine the components back into a complex number. Parameter split activations split the complex number into its magnitude and angle, apply a real function to them and then re-make the complex number with the new parameters. Fully complex activation functions aim to use only $\mathbb{C} \to \mathbb{C}$ functions.

A big concern when making a $\mathbb{C} \to \mathbb{C}$ mapping is whether it is holomorphic, or in other words, differentiable around certain points, and whether it is an *entire function*, meaning that the function is differentiable around every point [1].

Another concern is *Liouville's theorem* which states that all *bounded entire functions* are a constant, which means that any potential complex valued activation function will by default not be bounded and thus not allowing any activations that *squash* the range of the complex function into a smaller interval [1]. This problem, as of late, is not major, as many state of the art real valued networks do not use activation functions that *squash* intervals, instead opting for ReLU or ReLU like functions, which effectively just cut off part of the function's domain.

This section will provide activation functions that were examined and visualize their range where possible.

A. Hyperbolic Tangent

A staple activation function from the time before ReLU was popularized, since it is a hyperbolic function, a natural extension of it would be to move it into the complex domain.

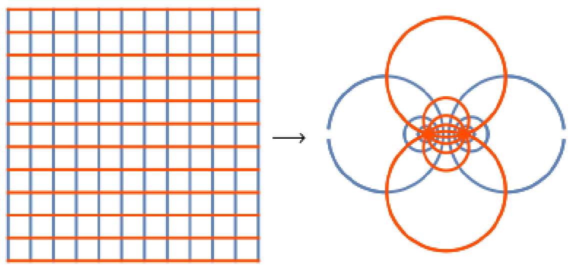

Fig. 1 Hyperbolic tangent mapping from a plane to circles

Hyperbolic tangent maps the complex plane into a series of circles in its range, thus making the space non-linear. The problem with such a function is much the same as in $\mathbb{R}$ it saturates quickly thus resulting in worse learning for very large models.

The function also suffers from many non-holomorphic points as seen in figures 2. and 3.

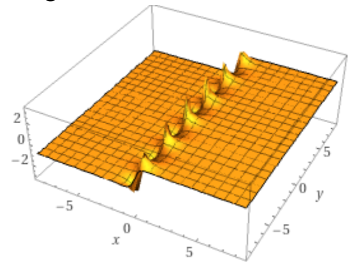

Fig. 2 Hyperbolic tangent real axis mapping

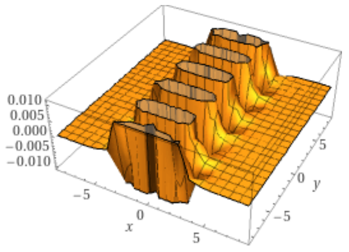

Fig. 3 Hyperbolic tangent imaginary axis mapping

B. ReLU

With the popularity of ReLU and ReLU like functions, a question was raised whether it works the same way for complex valued neural networks. The search for a ReLU-like complex function lead to the attempt of making ReLU map $\mathbb{C} \rightarrow \mathbb{C}$, using the following function:

$$\mathbb{C}ReLU(z) = ReLU(Re(z)) + iReLU(Im(z))$$

Resulting in the mapping shown in figures 4. and 5.

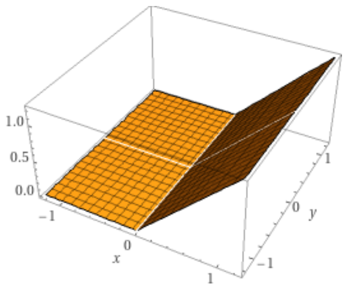

Fig. 4 Complex ReLU real axis mapping

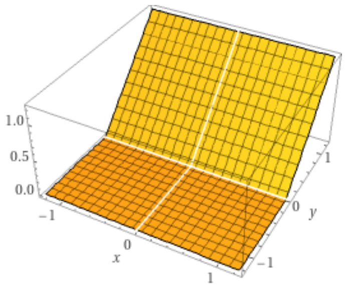

Fig. 5 Complex ReLU imaginary axis mapping

The mapping results in a range that looks much the same as its real valued counterpart, but the issue lies in what happens on the complex plane of the range. On a complex plain split into four equal quadrants, in the counterclockwise direction, the first quadrant of the domain would be mapped fully to the first quadrant of the range, the second quadrant would be mapped to the positive imaginary axis, the third would be mapped to $0 + 0i$, or just 0, and the fourth quadrant would end up being mapped to the positive real axis. This ends up squishing a whole quadrant to 0 which is a pole and complex functions usually don't act as expected near or on poles in the complex domain, and two quadrants to a single axis, basically deleting a component which could have contained hidden information about the point, leaving the whole domain with only a quarter of what it actually represented before being activated.

Due to this, complex ReLU was disregarded as detrimental to the actual learning process. It remains to be seen if ReLU-like functions that don't cut off most of the domain are viable.

C. Unit Activation

This complex activation function aims to do, in the complex plane, what ReLU does on the real line. ReLU cuts off all values below a certain threshold, to mimic that in the complex plane, a magnitude threshold is used. All values that are below magnitude 1 are brought down to 0, while all other values are left as they are.

This maps the complex plane into a new complex plane, with the unit circle from the original plane now mapping to 0. Thus this function does not accomplish anything other than punishing values with smaller magnitudes, which can then lead to the network forcing numbers to grow larger during the learning process.

This activation was also disregarded as being detrimental to the learning process.

D. Linear Scaled

The last function that will be presented is the linear scaled function. Even though *linear* is in its name, it makes the space non-linear, but the way it scales values is linear. It does this using the following function:

$$LinSca(z) = \frac{z}{r(z)}$$

This will force the values of the function to stay within the magnitude of 1, but leave the angle unchanged. This has the effect of implicitly making all complex values along a certain angle be equal, which eases the process of learning and helps with the classification method that will be discussed later.

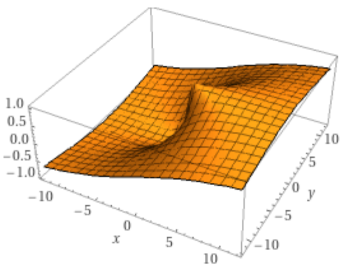

Fig. 6 Linear scaled function real axis mapping

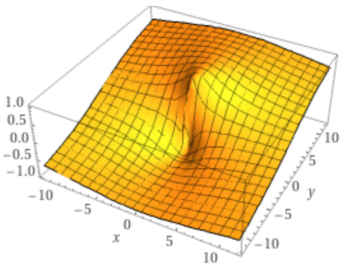

Fig. 7 Linear scaled function imaginary axis mapping

The linear scaled function has a singularity in 0, otherwise it is holomorphic across the whole domain, with the derivative being just $\frac{1}{r(z)}$. With this in mind the author recommends this activation function as they believe it best represents what ReLU does on the real line, and that is to limit the range to only certain values.

IV. Optimization

Again, as with linear algebra, for the purposes of this paper, the optimization that is used to learn parameters in real valued neural networks is directly translatable to complex valued neural networks without loss of generality.

Gradient descent is used to learn the many parameters that the neural network has. Gradient descent is an iterative optimization algorithm that converges towards a local minimum over iterations of updating parameters used in the optimization. The general form of gradient descent is the following:

$$x_n = x_{n-1} - \eta \nabla f(x_{n-1})$$

Meaning that the value of a function parameter x in the current iteration is equal to the result of subtracting the previous value of the parameter with the value of the gradient of the function at that point. The gradient is scaled down with a coefficient so that it isn't too large thus making the new parameter overshoot its local extremum. The gradient is a vector that points towards the direction in which a function grows, so in the case of finding the minimum of a function, subtraction is used to flip the vector to point towards where the function locally falls.

The value of the gradient of a function at some point is represented by the derivative of that function at that point, thus raising the need for functions that are optimized to be differentiable at least in the interval of optimization. Since the complex valued neural network model will have to behave the same as its real value counterpart, all the functions used inside the complex valued model have to be at least holomorphic inside the optimization interval.

An interesting fact with having the learn rate constant be in the complex domain is that the designer of the neural network model can place a complex valued learn rate. If during training the designer sees that the complex component of the loss grows more than the real component, then the designer can opt to lower the complex component of the learn rate to incentivise slower descent on the complex side, and vice versa.

V. Input and Output Representations

Each neural network is made up of neurons that form layers. In the case of deep neural networks, the input to the network is a vector representing an object or point in some $\mathbb{R}^n$ space, or in this case $\mathbb{C}^n$ space, where n represents the feature space size, or in other words the number of neurons in the input layer of the neural network. Almost always, though, there is more than one point in the dataset that is used for training, in that case a space $\mathbb{X}$ is of the size $\mathbb{X}^{m \times n}$, where m is the number of points in the dataset. Alongside the points, since this paper is focusing on supervised learning, without loss of generality, there is a set of labels that maps $1-1$ to the vectors in the input space. Each label represents the target that is expected on the output layer of the neural network. The job of the neural network is to then learn the best way to generalize the inputs to their respective outputs.

Hidden layers are also composed of neurons which together represent a vector that now exists in a new featurespace that usually has a different number of neurons than that of the previous layer. They serve the purpose of composing multiple nonlinear functions on top of the starting one to best generalize the distribution of the starting data.

The last layer of neurons in a neural network represents the output layer. This layer's neurons are then used for classifying the input object in the case of classification problems, or as a vector that predicts some value in the case of regression problems. This paper will focus solely on classification problems, as focusing on either or both will not lose any generality, since both have a target that the network needs to approach in the output layer.

This paper examined multiple ways of changing the output layer to better suit the added bonuses of working with the complex plane. In all cases, instead of having a vector be the output, the output was reduced to a single neuron, or in other words, a single complex number. All approaches revolved around splitting the complex plane into equal parts, seeing where the output of the network lies on the complex plane, and then classifying based on that.

In all approaches the complex plane was divided into equal segments of the unit circle centered around the 0 pole. The number of segments equals the number of unique classes that exist in the set of labels. The labels are then mapped to be at the center of each of those line segments of the unit circle, with magnitude 1 and angle between the edges of the interval of the line segment.

The first approach tries to classify the complex number based on its angle, the second approach classifies the complex number based on the distance to each of the line segment intervals' edges. The last approach classifies the complex number based on the distance to the center of each of the line segment intervals, which is the label itself.

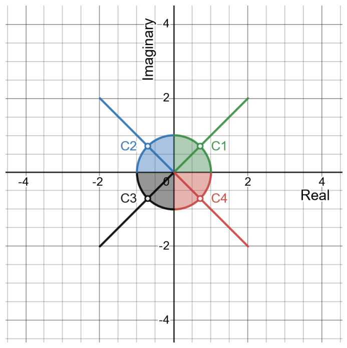

Fig. 8 Example of the output complex plane when classifying four classes. Each classification area has an area equal to a quarter of the unit circle. The labels, presented as points C1, C2, C3 and C4, are placed at the centers of the unit circle segment that belongs to that class. Lines present in the plot represent all the points that are equal to their respective labels based on angular distance. The lines equate to the first approach, the areas to the second approach, and the points to the third approach.

VI. Complex Valued Loss Functions

A very important part of modeling a neural network is the choice of the loss function, also called cost or error function. Real valued machine learning models have a variety of loss functions that are used in them, some of the more noteworthy are the Hinge function, Mean Squared Error (MSE) and the Cross Entropy Loss (CEL) function. Each of them have their own respective use for the tasks that they were designed to solve, for example MSE was designed for regression tasks, Hinge for boundary maximization and CEL for classification.

This paper explores some of the possibilities for fully complex valued loss functions, how they behave and in which cases they should be used.

As in section III, this section will outline some examined loss functions and visualize them where possible.

A. Argument Distance Loss

This loss function uses the angular distance function as its basis for calculating the loss. The function for angular distance is as follows:

$$\delta(\theta_1, \theta_2) = min(|\theta_1 - \theta_2|, 2\pi - |\theta_1 - \theta_2|)$$

The loss itself is calculated using this function:

$$\mathcal{L} = \sum_{i=0}^{m} \delta(\varphi(\hat{y}_i), \varphi(y_i))$$

For this loss function, all complex numbers along a line that has an angle φ are the same complex number. This approach is tailored for the first approach to classification presented in the previous section.

B. Interval Distance Loss

The interval distance loss function calculates how far the output of the neural network is to its corresponding label's classification area edges, taking the minimum of the two. It also uses angular distance as presented in the previous subsection, to have all the numbers along the same angle be the same number. When the point landed in the zone the function value would be set to 0. For classification area edges φ_1 and φ_2, the function can be calculated with:

$$\mathcal{L}(\hat{y}) = \begin{cases} 0, & \varphi_1 \leq \varphi(\hat{y}) \leq \varphi_2 \\ min(\delta(\varphi(\hat{y}), \varphi_1), \delta(\varphi(\hat{y}), \varphi_2)), & \text{else} \end{cases}$$

This function tries to emulate what the Hinge function does in the real domain, by forcing points to be within a certain area, no matter where in that area, but punishing points that are outside the area.

C. Complex Mean Square Error

As the name implies this is the direct translation of the real valued loss MSE to the complex domain, mapping $\mathbb{C} \to \mathbb{C}$, since Euclidean distance still stands in the complex domain, because it is isomorphic to the $\mathbb{R}^2$ vector space, the same function can be used in $\mathbb{C}$.

This loss function works best when the label is set to a specific complex number, like in the third approach to classification.

Since the Complex MSE maps $\mathbb{C} \to \mathbb{C}$ the resulting loss value is a complex number, thus the loss vs epochs plot is in the complex plane. There are two ways to then present such data, one would be to plot the full three dimensions with a single point on such a plot being denoted with the Re, Im and the epoch value of the loss. The second way to plot the loss is to plot the complex plane and color-code each of the points in the plane with a color gradient whose intensity depends on the epoch.

The characteristic loss curve that is present when using real valued loss is still present using complex valued loss functions. Except the curve follows a more pronounced path, akin to that of rolling a ball sideways down a bowl eventually slowing down and stopping in one of the local extremums.

VII. Implementation

The implementation of the Complex Valued Deep Neural Network Module (CVNN Module) consists of multiple submodules. This section will give detailed explanations of each of the submodules present within the module.

The implementation was done using the python programming language, version 3.11.5. All additional python modules, along with their specific versions, used within the implementation are available in the requirements file.

A. GPU and CPU Optimizations

Within the wrapped_numpy submodule, as the name implies, are wrapped numpy operations, literals and array manipulation functions. These functions serve the purpose of optimizing the execution time of the training, evaluating and using the CVNN Module. Functions wrapped include basic mathematical operations, summation, transposition, exponentiation, etc. Literal wrapping consisted of wrapping the numpy random uniform and random normal functions and zeros and empty like functions. For array manipulation functions, wrapped operations include, concatenation, finding unique values, the where function, as well as the real and imag functions.

The module automatically detects whether to use CPU or GPU optimization based on the fact if there is a present NVIDIA GPU.

CPU optimizations were done using the Numba python module. To further speed up the execution time, for Intel CPUs, it is highly recommended to additionally install the icc-rt module that will further optimize code for Intel CPU instructions. As of the time of writing this paper, all the functions needed for the CVNN Module were able to be wrapped by annotating them with *@njit* and writing the adequate numpy function. For some of the functions, only some derivatives of them were available, examples include summation, where either summation along a specific axis is available or the summation of a whole object numpy ArrayLike is available, matrix multiplication, for which only the single argument form is available. Where applicable, other than the *nopython* flag (denoted as *n* in the annotation of a function), more just in time compiler flags were used, such as *fastmath* for mathematical operations.

For GPU optimization, numpy wasn't wrapped, but instead the CuPy module for CUDA programming was used. The CuPy module comes with all the functions needed for the CVNN module already built in, therefore the optimization was straightforward calls to those functions under new names, to match those wrapped using Numba.

B. Utility functions

Within the utilities module are functions that are used throughout the CVNN module, that help in any way needed.

Within the synthetic data file are blueprints for making binary and multiclass classification task datasets. The dataset utilities file contains functions for minibatching, normalizing and shuffling data. The plotting file contains functions for plotting real and complex valued losses, and plotting of a real valued metric. Prediction utils file contains functions for transforming arguments of complex numbers to categorical singular values and a function to transform a complex valued vector into a category for classification purposes. The label utils file contains functions to map categorical label values to one of the three approaches of classifications explained in section V.

C. Activations

This module contains classes for all the available activation functions. Currently available are the complex hyperbolic tangent, linear scaled activation and unit activation. These were left as possible candidates for $\mathbb{C} \to \mathbb{C}$ activation functions, after others were ruled out, as per section III.

This module also contains a dictionary that maps these activations to their names represented as strings, for later ease of use.

Each new activation class is required to conform to the Activation base class which has two methods that need to be implemented, activate and deactivate. The activate method performs the activation, whilst the deactivate method computes the derivative of the activation.

The Activation base class can be used for addition of new activation functions, as long as the new function conforms to the interface left by the base class.

D. Losses

As with activations, this module contains the classes for each of the loss functions discussed in the previous section. Alongside it is the base Loss class and the dictionary for string names of losses.

To conform to the Loss base class, the calculate loss and loss gradient methods have to be implemented. The calculate loss method computes the loss value given some predictions and labels, whilst the loss gradient computes the derivative of the loss function with respect to the predictions.

Although all the discussed loss functions are available in the module, the module only handles Complex MSE, this is a fact that will be discussed more in detail in the *Results and Discussion* section, but in short, it has shown that it performs the best for learning purposes.

E. Metrics

The metrics module houses all the available metrics that can be used for model evaluation. All the metrics can be used during training as well. It also contains a dictionary that gives a string name to each of the metrics for ease of use later. Inside the module are also metric utilities, used for getting all the needed information for later metric calculation, information such as true positives, false positives and so on, all calculated per class to allow metric use in multiclass classification tasks and give the user an option for macro and micro metrics on demand without the need to implement anything additionally.

Within the utils file are the functions get correct, instances and all classified, each working per class, and a function to wrap those three into a dictionary. Get correct returns the number of predictions and labels that are equal to the passed class at the same time, get instances returns the number of times the passed class is found within the labels, while get all classifications returns the number of times the passed class was found in the predictions.

All metrics are derived from the Metrics base class which only requires the calculate metric method to be implemented. If the metric needs any additional parameters they can be added in the initialization method of the class, same goes for stateful behavior of the metric if needed. Addition of new metrics only needs to implement the Metric base class.

F. *Optimizers*

The optimizers module contains optimization methods that can be used for learning during model training. The module offers gradient descent (GD) and ADAM optimization. Whether stochastic, mini-batch or regular GD is used depends on the size of batches inputted into the model when the model is created. In this module is also a dictionary that offers string names for optimizers.

To implement an optimizer, the Optimizer base class needs to be implemented. It is highly recommended that if the new optimizer needs any parameters for optimization, i.e. learn rate, epsilon normalization factor, previous gradients etc. to add them in the initialization method of the class. The Optimizer base class also contains an optimizable layers field, which is a field with layers that have parameters that can be optimized by the optimizer. The build method is used to initialize any other parameters that couldn't be initialized in the initialization method, like the shapes of previous gradients for optimizable parameters. After that is the update parameters method that does the optimization on all the optimizable layers' parameters.

G. *History*

History is a stateful class within the history module that contains all the values of validation and training losses and metrics throughout the training process of the model. It stores all the values in a dictionary field called history, which has the names of all the metrics used in the model, as well as the loss. Training keys are prefixed with *"train"* while validation keys are prefixed with *"val"*.

The class also offers state updates, getting the current state and plotting either loss or metrics up to the current state.

H. *Layers*

This module currently only houses the implementation of the dense, fully connected layer called just Layer.

The Layer class contains all the parameters that it needs to operate, input shape, number of neurons in it, weights and biases, their gradients, the input, transfer and output values, the activation function, and a name.

The build method initializes all the parameters at model runtime, based either on the input shape or number of neurons in the layer before this one. Other methods include transfer which applies the transfer function, activate which activates the transferred values, forward which wraps them, and backward which deactivates the received input and computes all the gradients.

I. *Model*

The class Model class, housed inside the model module, contains the implementation of a neural network model. To make a model object a list of layers is needed. Inside the model are fields for the layers, a history object, optimizer, loss and metrics.

Akin to compiling the network in Tensorflow, building a model in the CVNN Module requires the user to pass an optimizer, the loss function and a list of metrics to be used, each of which is then built using their own build methods if applicable.

The model offers public methods for computing loss given some labels and predictions, computing metrics given, again, labels and predictions, testing the model given some input and labels which returns a dictionary with the calculated loss and metrics for passed input and labels and a predict method that returns complex values which represent the output of the network given the passed data. Lastly, and most importantly, a train method, which trains the network. To call the train method the user needs to pass training and validation data and labels, the batch size and the number of epochs for which to train the network. The network will then update the history field which can then be accessed afterwards to verify results. During training the network provides a progress bar with training and validation loss values for the current epoch for online tracking of training progress.

VIII. Evaluation

The CVNN Module was evaluated on multiple tests, ranging from comparison with a real valued deep neural network (DNN), to evaluating the effectiveness of different losses and activations.

Tasks which were used for evaluations were a binary classification task, consisting of 1000 points in 5 dimensional complex vector space. Each class consists of a gaussian distribution cluster, one class centered around $+5+5i$, and the other around $-5-5i$, with deviation fixed to 1. The other task is a multiclass classification task, consisting of 5 classes with a total of 1000 points of data. Each class' cluster center was placed radially around a scaled unit circle of radius 7, then around that center a gaussian distribution was applied with deviation 1. Like in the binary classification task, the featurespace was 5 dimensions. When creating the datasets, the random seeding number was fixed.

For evaluating the effectiveness of losses, multiple models were created with the same architecture and optimizer, but with different losses, each was given the same binary classification task, then it was evaluated on the accuracy, F1, precision and recall metrics.

Evaluation of activations was done on the same task as loss evaluation. This time with fixed layer number, neuron number, optimizer and loss function. Also tested were different permutations of activation functions in the layers. The models were evaluated on the same metrics as losses.

Lastly, for comparison with DNNs made using the Tensorflow and Keras APIs, metrics used for comparison were macro F1 score and accuracy. For both models a minimal architecture was found that was able to adequately learn and generalize the task that was given. The models were compared on the number of learnable parameters needed, size of the architecture, iterations needed to reach a local extremum and the values of the aforementioned metrics. Fixed parameters were the optimizer and batch size. The optimizer was set to ADAM and batch size to 64.

IX. Results

This section will go over the results of the tests that were defined in the evaluation section.

A. Loss evaluation

The results of evaluating the three losses discussed earlier in the paper showed that two of the losses did not have the neural network learn the correct parameters and generalize well on the problem. Even with reduction of the learn rate the losses failed to generalize the given data on a trivial problem.

The Complex MSE was the only one able to achieve wanted results, even showing overfitting in some tests, whilst Argument Distance and Interval Distance failed to learn even the simple binary classification problem.

B. Activation evaluation

Activations, much like loss evaluation, only provided a single activation function that worked on the binary classification problem. Linear scaled activation was the only activation that managed to learn and generalize well on the binary classification task, and only networks with just this activation were able to do so. The second closest permutation was with sporadic uses of complex hyperbolic tangent, which as predicted usually ended saturated much of the time, even on normalized data. The unit activation fared even worse with the network not being able to learn anything with it as an activation function.

C. CVNN vs DNN

On the binary classification problem, both networks learned and generalized the data well, with minimal overfitting for both, and with accuracy and F1 reaching 0.99. The number of input neurons for the CVNN was 5 while that number was 10 for the DNN, the number of neurons in the first and only hidden layer was 3 for both, while the number of output neurons was 1 for the CVNN and 2 for the DNN. Thus meaning that the overall number of parameters was smaller in the CVNN by a factor of 2. The number of iterations needed to train both the networks to an adequate level was less for the DNN than for the CVNN, being 100 and 256 respectively, but due to the much smaller size of the network the difference isn't too large.

For the multiclass classification problem, both networks suffered from randomness of the initialization of their parameters, sometimes they would reach a good local minimum in which they learn and generalize well, and sometimes that would not happen, other times the DNN would completely overfit, with the DNN being more resilient to the randomness than the CVNN.

Another problem for CVNN modeling was calculating the correct learn rate, as it would often reach a good local minimum and then leave it for a worse one during the learning process.

The DNN's highest metrics, when not overfitted, were 0.95 for both accuracy and F1, when not overfitted. The DNN's architecture consisted of three layers, each with 5 neurons, for the sum total of 75 parameters. The CVNN eventually reached the same scores as the DNN with again, less parameters, this time by a factor of 2.5, as the architecture needed just one output neuron and half the input neurons. The iterations stayed at the same ratio.

X. Conclusion

The CVNN has proven to be on par with the DNN on the problems of multiclass and binary classification, with the ratio between the number of parameters rising as the size of the DNN enlarges. A downside to the CVNN is the fact that it is harder to input the correct hyperparameters, like learn rate when modeling, but this can be overcome as more CVNNs are made. The CVNN also has somewhat slower learning, but that is a fact that is left to be proven whether it is true, because of the novelty of modeling such networks.

The proposed CVNN also utilizes the complex plane to its fullest when classifying, only needing a single output neuron and simple activation functions to learn and generalize a problem.

The space to advance activation functions, optimization methods, loss functions and architectures in CVNNs is undeniably large, this paper showed what just a small amount of modifications from the real world over to the complex one can improve.

References

1. Visual Complex Analysis, Tristan Needham
2. Bassey, J., Qian, L. and Li, X., 2021. A survey of complex-valued neural networks. *arXiv preprint arXiv:2101.12249*.
3. Hirose, A. and Yoshida, S., 2012. Generalization characteristics of complex-valued feedforward neural networks in relation to signal coherence. IEEE Transactions on Neural Networks and learning systems, 23(4), pp.541-551.
4. Yi, Q., Xiao, L., Zhang, Y., Liao, B., Ding, L. and Peng, H., 2018, November. Nonlinearly activated complex-valued gradient neural network for complex matrix inversion. In *2018 Ninth International Conference on Intelligent Control and Information Processing (ICICIP)* (pp. 44-48). IEEE.
5. Georgiou, G.M. and Koutsougeras, C., 1992. Complex domain backpropagation. *IEEE transactions on Circuits and systems II: analog and digital signal processing*, *39*(5), pp.330-334.
6. Scardapane, S., Van Vaerenbergh, S., Hussain, A. and Uncini, A., 2018. Complex-valued neural networks with nonparametric activation functions. *IEEE Transactions on Emerging Topics in Computational Intelligence*, *4*(2), pp.140-150.

7. Kobayashi, M., 2019. Noise robust projection rule for hyperbolic Hopfield neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, *31*(1), pp.352-356.
8. Popa, C.A., 2017, May. Complex-valued convolutional neural networks for real-valued image classification. In *2017 International Joint Conference on Neural Networks (IJCNN)* (pp. 816-822). IEEE.
9. Kobayashi, M., 2019. $ O (2) $-Valued Hopfield Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*, *30*(12), pp.3833-3838.
10. Ding, L., Xiao, L., Zhou, K., Lan, Y., Zhang, Y. and Li, J., 2019. An improved complex-valued recurrent neural network model for time-varying complex-valued Sylvester equation. *IEEE Access*, *7*, pp.19291-19302.
11. Scardapane, S., Van Vaerenbergh, S., Hussain, A. and Uncini, A., 2018. Complex-valued neural networks with nonparametric activation functions. *IEEE Transactions on Emerging Topics in Computational Intelligence*, *4*(2), pp.140-150.
12. Marseet, A. and Sahin, F., 2017, November. Application of complex-valued convolutional neural network for next generation wireless networks. In *2017 IEEE Western New York Image and Signal Processing Workshop (WNYISPW)* (pp. 1-5). IEEE.
13. Wilmanski, M., Kreucher, C. and Hero, A., 2016, December. Complex input convolutional neural networks for wide angle SAR ATR. In *2016 IEEE Global Conference on Signal and Information Processing (GlobalSIP)* (pp. 1037-1041). IEEE.