Stochastic for material science

Programming assignment

TECHNISCHE UNIVERSITÄT
BERGAKADEMIE FREIBERG

Die Ressourcenuniversität. Seit 1765.

SOLVING DIFFERENTIAL EQUATIONS USING NEURAL NETWORKS (AUTODIFF)

Viswambhar Reddy Yasa

Computational Material Science

65074

Supervised by
Dr. Felix Balani

August 20, 2021

Contents

1 Introduction

The previous decade saw an unprecedented growth in the application of Machine and Deep learning techniques in almost all fields of science,technology,finance etc supplemented by the change in ideology from "Think and Store Data" to "Store Data and Think".

Machine learning discovers rules to execute a data-processing task, given examples of what's expected.

-Francois Chollet[1]

Mostly, Machine learning is used as advance curve fitting tool, i.e We fit a curve amongst the known data points to know hitherto unknown data points ,only now using some sophisticated tools.There is also another characteristic of Machine learning, which in my view is far less utilized than for curve-fit applications and that is the ability to discover the underlying rules of the data which is used to train. This project tries to utilize this aspect of machine learning and deep learning, especially Neural Networks.The broad question or the leitmotif would be:

Is it possible/How to develop Machine learning/deep learning models which do not have any prior information about the output data(no training data) but know how the data behaves(rules of the underlying data) to predict the output?

In this project, the models used are Long-Short term Memory Neural Networks(LSTM) and the rules are given in the form of differential equations with sufficient boundary conditions and therefore the output will be the solution of the differential equation and thus the title, *Solving differential equations using Neural Networks.*

This is done by:

1. Developing an Automatic differentiation framework
2. Building the Neural network architecture and optimizers to train the models using this framework
3. Approximating the solution as a Neural Network by formulating the loss function according to the differential equation along with boundary conditions and obtaining the solution
4. Extending the proposed method as surrogate model to classical Runge-Kutta stepping schemes for solving more complicated differential equations.

2 Theory

The basic theme of the whole field of Machine learning is the following : A computer is faced with a task and an associated performance measure, and its goal is to improve its performance in this task with experience which comes in the form of examples and data.[2]

2.1 Neural Networks

Neural Networks are a specific type of models which are somewhat inspired by working of human brain. Small blocks of information(called *Neurons*) fire in a specific manner(according to the *activation function*) with a specific intensity(called *weights*) in a specified sequence which will capture the behaviour of complex data by optimizing the weights(by using *optimizers*) to minimize the *loss function* constructed using the training data.

Types of Neural Networks:

Information passes from one set of neurons(called a *layer* which can be thought of as single unit of computation) to another set of neurons. Based on these interconnections between the layers and type of the individual neuron several architectures of Neural Networks are developed. Many of such architectures are given in figure 1

Activation functions:

Activation function characterizes the behaviour of a neuron. It is a function which manipulates the incoming data into the neuron to produce a output which can be the final output or input to another layer. The activation functions channelize the flow of data leading from input to output via hidden layers. Some typical activation functions used in Neural Networks are given in the figure 2 along with corresponding formulae.

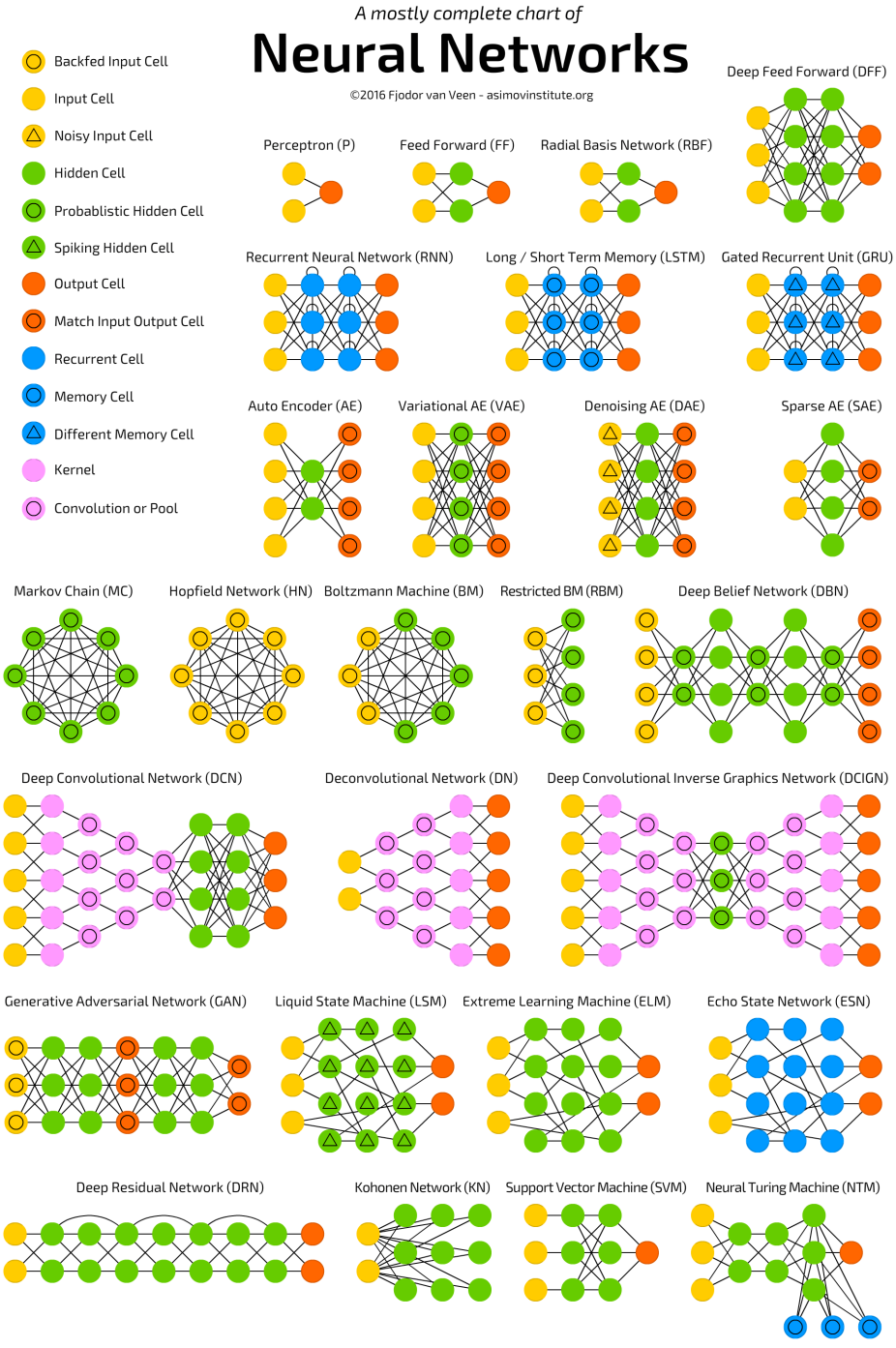


Figure 1: Neural Network Architectures[3]

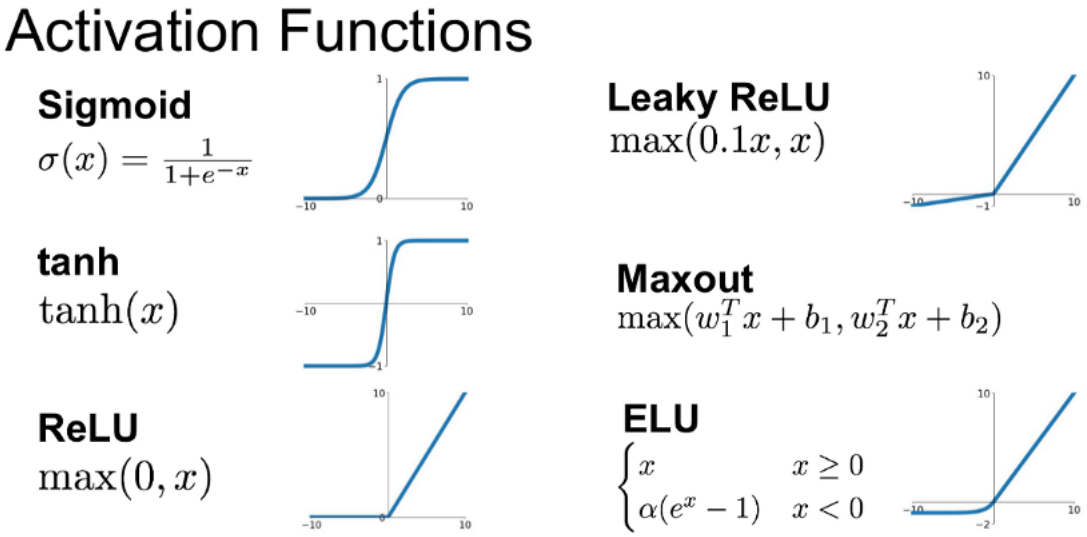


Figure 2: Activation functions[4]

A simple Neural Network:

Based on the above definitions , a simple Neural Network , generally called as an Artificial Neural Network(ANN) or a Multi-layer perceptron along with it's mathematical formulation is shown in the figure 3.

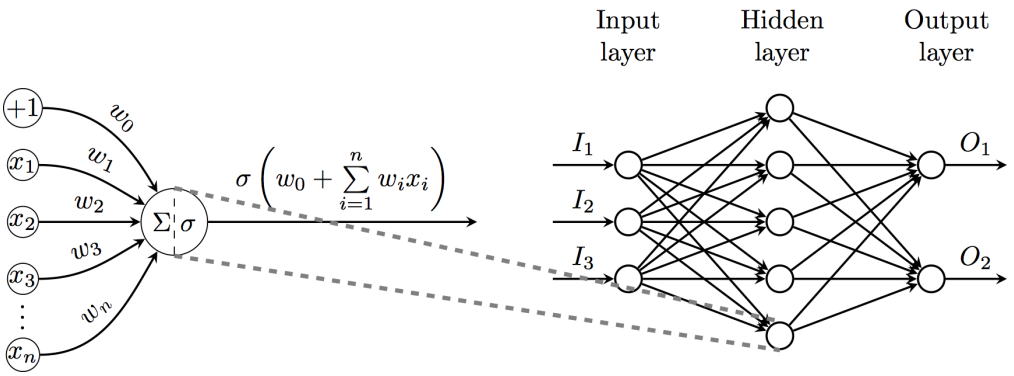


Figure 3: A simple Neural Network[5]

Loss function:

Loss function is the performance measure in the Neural Networks. Weights are altered by the optimization algorithms to minimize this loss functional. Generally, the loss function penalizes the output of the Neural network

against the known training data points.
General mathematical formulation is as follows, Let the training data (known outputs for inputs) be given as input-output pairs as :

$$\{(x_1, y_1), \ldots, (x_N, y_N)\} \subseteq X \times y \tag{1}$$

Where X is the input sample space and Y is the output sample space.
The goal of a Neural Network(or any ML model for that matter) is to find a function which maps from X to Y and gives predictions($\hat{y}$) according to training data, i.e 2

$$f : x \mapsto \hat{y} \tag{2}$$

This calculation of f is sometimes called "Forward propagation"(as in we are moving forward direction in a Neural Network) and computing the loss and gradients is referred as "Backward propagation"(as in we are moving in the reverse direction in Neural Network calculating the error). As a performance measure(to see whether f is doing a good job) loss function is defined as 3:

$$L : Y \times y \to R \tag{3}$$

Here L measures how close actual y is to the predicted output $\hat{y}$

1. squared loss:
$$L(y, \hat{y}) = |y - \hat{y}|^2 \text{ for } y = R$$
2. zero one loss:
$$L(y, \hat{y}) = 1_y(\hat{y}) \text{ for arbitrary } y$$
3. cross-entropy loss:
$$L(y, \hat{y}) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})) \text{ for } y = [0, 1]$$

Optimizers:

Weights of a Neural Network need to be changed to decrease the loss function. Optimizers govern how these weights are changed to get optimum set of weights which minimize the loss. *Gradient based optimizers* are based on the fact that the gradient of a function gives the direction of it's decrease. The general theme of all gradient-based optimizers is : Compute the gradients and manipulate these gradients(e.g multiply with negative of learning rate) and add them to present set of parameters. How do we manipulate the gradients to make them more efficient forms the factor for different types of algorithms. This is elucidated in the figure 4 specific optimizers built in this framework are discussed in later chapters.

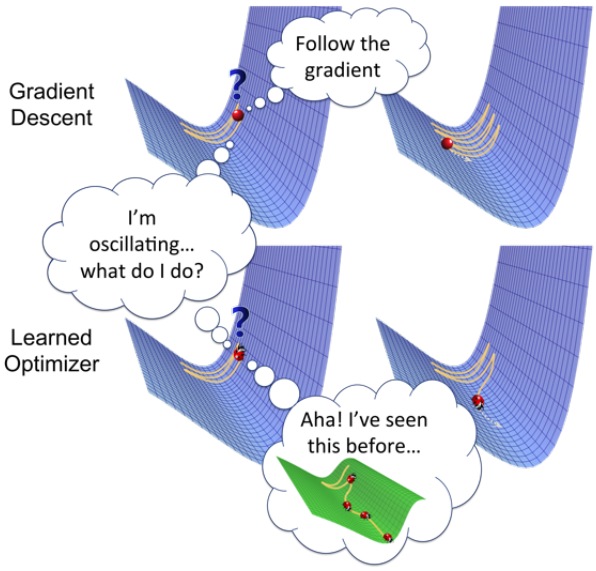


Figure 4: General theme of Gradient based Optimizers [6]

LSTM Neural Networks: This is a type of Neural Network that will be particularly used in this project and hence a small outline is provided here. Neural Networks ,when training for particularly large problems are prone to this problem called *Vanishing Gradients*. To put it in simple terms, it means that the Neural Network doesn't remember the information that is being passed in the backpropagation in the form of gradients. LSTM Networks are designed to solve this problem. This is done by employing a characteristic called *memory* to each neuron(now called LSTM units) to remember the relevant information passed on from previous layers. Each LSTM unit contains an input , forget and memory gates to regulate the flow of information. The magnitude of the gradients passed is also learned by the network as a parameter during training and due to this the memory aspect comes into play. It is illustrated in the figure 5

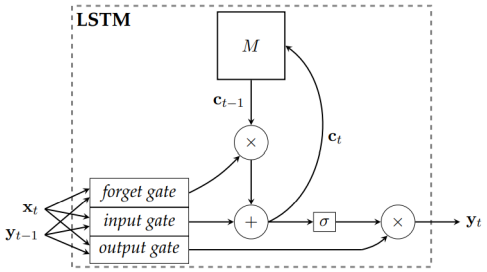


Figure 5: Architecture of general LSTM Unit . Source:"A trip down long-short memory lane" by Peter Velickovic (https://www.cl.cam.ac.uk/~pv273/slides/ LSTMslides.pdf)

2.2 Differential Equations

This section deals with customary yet brief information about differential equations.
Differential Equations are prevalent and ubiquitous in science and technology. They are used to model various types of phenomenon around us.A general differential equation establishes relationship between a uni/multivariable function with its derivatives(ordinary or partial).
There is a great deal of variety in the types of differential equations that one can encounter both in terms of form and complexity. They can vary in *order*; they may be *linear* or *nonlinear*; they can involve various types of *initial/terminal conditions* and *boundary conditions*. In some cases, we can encounter systems of coupled PDEs where multiple functions are connected to one another through their partial derivatives. In other cases, we find free boundary problems or variational inequalities where both the function and its domain are unknown and both must be solved for simultaneously.
The differential equations which are solved under this framework are discussed in later chapters.

2.3 Theory of solving differential equations using Neural Networks

Beginnings and some literature: The idea of solving differential equations by using Neural Networks was first introduced in 1996 by *Isaac Lagaris, Aristidis Likas Dimitrios Fotiadis* [7] . The basic idea was based on the *The Universal Approximation Theorem* of Neural Networks .This led to the development of deep learning models which don't use any data for training but

use the underlying physics. This method was greatly improved by the advent of high-performing computers equipped with Automatic differentiation frameworks like Tensorflow, Keras , PyTorch etc. Using the same principle, models called *Physics Informed Neural Networks*(PINNs) have been developed to solve the equations with no prior data about the solution and also with sparse data using classic Runge-Kutta stepping schemes for high dimensional problems [8]. The same principle was developed as *Deep Galerkin Method*(DGM),which uses a deep neural network instead of a linear combination of basis functions. The deep neural network is trained to satisfy the differential operator, initial condition, and boundary conditions using stochastic gradient descent at randomly sampled spatial points [9].

Universal Approximation Theorem of Neural Networks[10] : This is a theorem that gives mathematical proof of why this method works and How does a Neural Network approximate as something with no idea of output data. The simple crux of this theorem is that any continuous function defined in a compact subset can be approximated by a Neural Network with one single hidden layer, irrespective of the type of activation function.

Mathematically, the statement of the theorem is as follows: let ϕ be a nonconstant, bounded, monotonically-increasing continuous function and let I_m denote the $m-$ dimensional unit hypercube. Then, given any $\epsilon > 0$ and any function f defined on I_m, there exists $N, v_i, b_i, \mathbf{w}$ such that the approximation function

$$F(\boldsymbol{x}) = \sum_{i=1}^{N} v_i \phi \left(\mathbf{w} \cdot \boldsymbol{x} + \mathbf{b}_i \right)$$

satisfies $|F(\boldsymbol{x}) - f(\boldsymbol{x})| < \epsilon$ for all $\boldsymbol{x} \in I_m$ [2]. A small word of caution is necessary here. This theorem proves only the existence of a solution, it doesn't speak about feasibility of finding the parameters like number of neurons, weights , number of hidden layers.

Basic concept and algorithm for solving a Differential Equation : The general form of any differential equation along with initial and boundary conditions is given in equation 4 [9]

$$\begin{aligned} &\tfrac{\partial u}{\partial t}(t,x) + \mathcal{L}u(t,x) = 0, \quad (t,x) \in [0,T] \times \Omega \\ &u(t=0,x) = u_0(x) \\ &u(t,x) = g(t,x), \quad x \in \partial\Omega \end{aligned} \tag{4}$$

Goal is to approximate the solution of differential equation *u(x,t)* with a neural network f(t, $\boldsymbol{x}$; $\boldsymbol{\theta}$). To do this a loss function is constructed as a measure of how well the approximation satisfies the differential operator. It looks something like equation 5 [2]

$$\left\|\left(\partial_t + \mathcal{L}\right) f(t, \boldsymbol{x}; \boldsymbol{\theta})\right\|^2_{[0,T]\times\Omega} \tag{5}$$

Now boundary conditions need to be satisfied to get meaningful solution this can be done in two ways:Hard assignment of boundary conditions Soft assignment of boundary conditions

Hard assignment of boundary conditions: Hard assignment is forming a trial solution beforehand such that it satisfies all the boundary conditions apriori.
Advantages: Boundary conditions are accurately satisfied,No need to additionally train on boundary regions.
Disadvantages: Only available for simple settings.

Soft assignment of boundary conditions: Soft assignment is adding more terms to loss function such that it also measures how well the boundary and initial conditions are satisfied. The whole loss function in this domain will look like equation 6 [2]

$$L(\boldsymbol{\theta}) = \underbrace{\left\|\left(\partial_t + \mathcal{L}\right) f(t, \boldsymbol{x}; \boldsymbol{\theta})\right\|^2_{[0,T]\times\Omega}}_{\text{differential operator}} + \underbrace{\left\|f(t, \boldsymbol{x}; \boldsymbol{\theta}) - g(t, \boldsymbol{x})\right\|^2_{[0,T]\times\partial\Omega}}_{\text{boundary condition}} + \underbrace{\left\|f(0, \boldsymbol{x}; \boldsymbol{\theta}) - u_0(\boldsymbol{x})\right\|^2_{\Omega}}_{\text{initial condition}} \tag{6}$$

Advantages: Universal (almost every type of boundary condition can be easily incorporated), Elegant.
Disadvantages: More number of points need to be collected, loss function becomes complicated.
This loss function is constructed for a number of points in the domain and boundaries of the differential equation and that loss is minimized using any optimization techniques used in deep learning.Once the loss is minimized , the neural network model now approximates the solution of the differential equation within the domain of randomly sampled points.
Now the general algorithm to solve any differential equation is :

1. Set up the Neural Network model(number of hidden layers, neurons per hidden layer,output dimension,input dimension).If using hard assignment , form the trial solution.

2. Initialize the weights of the model

3. Setup an appropriate optimizer(number of parameters, learning rate and optimizer-specific parameters).

4. Randomly sample points along in domain in which the differential equation solution is to be found(Collocation).If using soft assignment collect random samples along boundaries also.

5. Construct loss function (hard/soft assignment) using the differential equation and these collected points.

6. Use optimizer to minimize this loss. Once the loss is minimum, the Neural Network model now approximates the solution of the differential equation within the domain in which random samples are collected.

2.4 Automatic Differentiation

Formulation of loss function and usage of gradient based optimizers reveal that we need derivatives of the model with respect to the inputs, parameters. The order of these derivatives is equal to the order of the differential equation .It is a painstaking and not very efficient method to write backpropagation formulae and applying chain rule at each node for each of the derivative and assemble them in the loss function. An overview of simplest form of backpropagation is elucidated in the figure 6

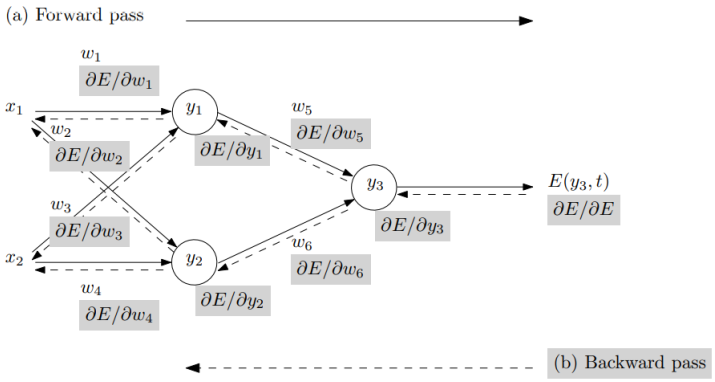


Figure 6: Overview of backpropagation. (a) Training inputs x_i are fed forward, generating corresponding activations y_i. An error E between the actual output y_3 and the target output t is computed. (b) The error adjoint is propagated backward, giving the gradient with respect to the weights $\nabla_{w_i} E = \left(\frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_6}\right)$, which is subsequently used in a gradient-descent procedure. The gradient with respect to inputs $\nabla_{x_i} E$ can be also computed in the same backward pass.
[11]

The derivatives of a function can be done using different approaches. They are shown in the figure 7. A compare and contrast among them is elucidated in

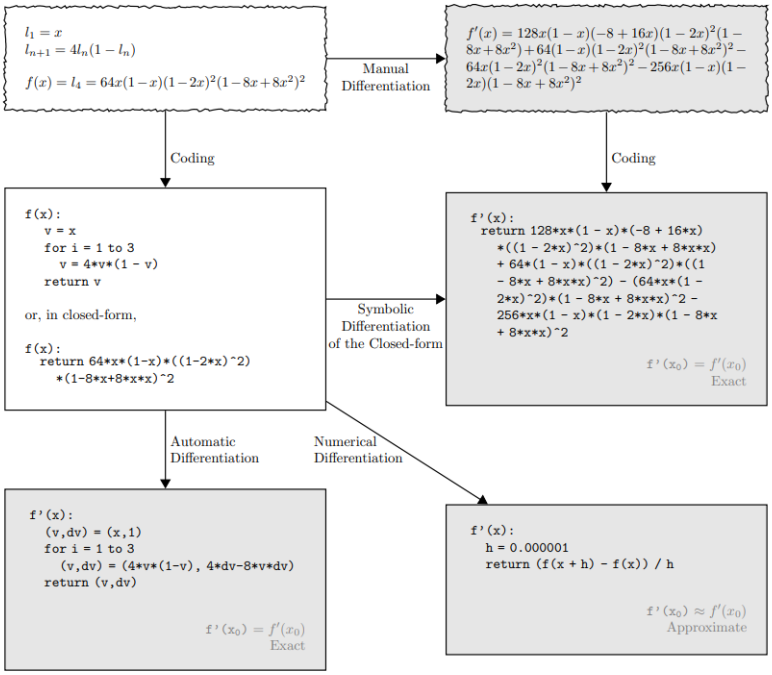


Figure 7: The range of approaches for differentiating mathematical expressions and computer code, looking at the example of a truncated logistic map (upper left). Symbolic differentiation (center right) gives exact results but requires closed-form input and suffers from expression swell; numerical differentiation (lower right) has problems of accuracy due to round-off and truncation errors; automatic differentiation (lower left) is as accurate as symbolic differentiation with only a constant factor of overhead and support for control flow. [11]

Automatic differentiation is the process of calculating derivatives by using *Computational Graphs*.It exploits the fact that no matter how complicated the function might appear, it is still comprised of primitive operations and manipulation of variables, which are generally multi-dimensional arrays(Tensors) whose derivatives can be collected without any effort using basic principles. A computational graph is defined as a directed graph where the nodes correspond to mathematical operations as a way of expressing and evaluating a mathematical expression.A simple computational graph is shown whose derivatives are accumulated in reverse in figure 8. If the derivatives are accumulated in forward direction it is called forward mode AD.

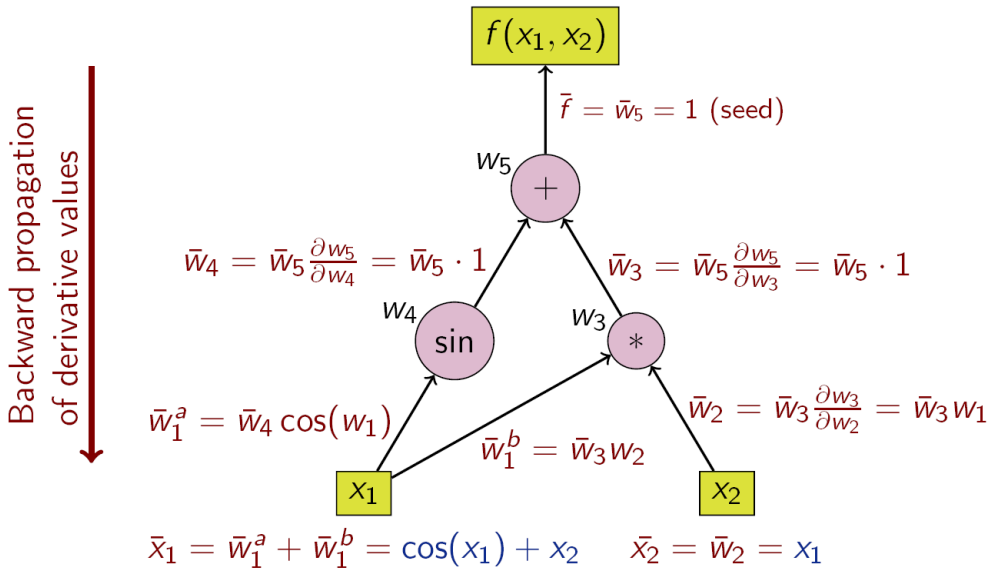


Figure 8: Calculating derivatives from a Computational Graph by Berland at English Wikipedia - Transferred from en.wikipedia to Commons by JRGomà., Public Domain, https://commons.wikimedia.org/w/index.php?curid=4860887

Some famous and widely used Automatic Differentiation packages are Tensorflow, Keras, Torch, Theano , Autograd.

3 Implementation Details

The programming of this project is done in *Python* the whole project can be broadly divided into three parts.

3.1 Implementation of Automatic Differentiation framework as a package

Outline: Taking derivatives symbolically or using numerical methods of Neural network models is not feasible, hence a framework of Automatic Differentiation(named *Autodiff*) is developed as a python package.
The aim of this Autodiff is to create a framework, which builds computational graphs and facilitates taking derivatives algorithmically by traversing over these graphs.On the outside , it should look like a normal Numpy operations but under the hood, computational graph should be drawn for every operation performed and derivatives should be collected accordingly so that when called for gradients, they can be accumulated effortlessly.This is what commercial packages like Tensorflow or PyTorch actually do.Hence this package can be thought as a simpler version of Tensorflow which uses Numpy as numerical kernel.

The skeleton for this framework is taken from an open source repository autodiff by Bruno Gavranović(https://github.com/bgavran/autodiff) and operations required are added to this skeleton to make this sufficient for problems to be solved using this framework.

Structure and Data: The basic data structure in autodiff is called a *Node* and every operation and Variable are just extensions(derived/child classes) of Node.
The Node class has the following attributes:

1. Name
2. list of children of the Node, i.e all the nodes which precede this node (if not a node, then a variable as starter of computational graph)
3. ID (a counter to trace easily along the graph)
4. context list to perform operations based on.
5. shape to get the shape from an operation done at the node or variable like a multi-dimensional array.

Basically there are two things that need to be done at every Node(Methods):

1. Calculate what is the value at the current node(eval)

2. Calculate what is the derivative at the current node.(partial derivatives)

Both these methods require the information about previous children, i.e value/gradient requires what is the value/gradient coming from children. If the Node is a variable(starter of a computational graph) then the previous gradient is simply array with the shape of the Node containing all ones.
Concepts of polymorphism and inheritance are applied as both these methods are implemented as virtual private methods(hence, the prefix underscore), meaning in the base class(node) no functionality is implemented and when called from base class object, they raise NotImplementedError and they are not inherited in the extended classes.Doing this makes higher order gradients available at no extra computational cost .Extended classes need to implement these two methods according to their functionality. This is enforced by defining two public methods of same name (eval and partial derivative, but no underscore) such that these methods are accessible by all the extended classes and they are used to call their private counterparts within the context. The Node class definition is given below.

```
class Node:
    """
    Node is like  the blue print for every operation and

    variable to be defined in the autodiff package
    Every Node does two things:
    1.Calculates the value
    2.Passes the information about gradient
    Therefore it is like a wrapper around Numpy which uses

    it as a Numerical Kernel.
    """
    #Some attributes common to all the NOdes
    epsilon = 1e-12
    id = 0
    context_list = []

    def __init__(self, children, name="Node"):
        """
        params:
```

```
        children : attributes which enter a node to be

        operated (eg: two numbers which need to be added)
        Node Id : to keep track of number of nodes

        instantiated to sort while collecting the derivatives
        cached: To check whether overriding has taken place
        shape: shape of the value of node if dealing with numpy arrays
        """
        # wraps normal numbers into Variables
        self.children = [child if isinstance(child, Node)

        else Variable(child) for child in children]
        self.name = name
        self.cached = None
        self.shape = None

        self.context_list = Node.context_list.copy()
        self.id = Node.id
        Node.id += 1

    def _eval(self):
        """
        This is left unimplemented and _ is used in the

        function name(Virtual method) such that:
        1.It is a private method (Not accessed by the

        object directly)
        2.It is overriden by all the child classes

        according to their operation (Add class adds two

        numbers and so on..)

        :return: returns the value of the evaluated Node
        """
        raise NotImplementedError()

    def _partial_derivative(self, wrt, previous_grad):
        """
```

```
        Method which calculates the partial derivative of

        self with respect to the wrt Node.
        By defining this method without evaluation of any

        nodes, higher-order gradients
        are available for free.

        This is left unimplemented and _ is used in the

        function name(Virtual method) such that:
        1.It is a private method (Not accessed by the

        object directly)
        2.It is overriden by all the child classes

        according to their operation (derivative of x*y

        wrt x is y and so on..)

        :param wrt: instance of Node, partial

        derivativative with respect to it
        :param previous_grad: gradient with respect to self
        :return: an instance of Node whose evaluation

        yields the partial derivative
        """
        raise NotImplementedError()

    def eval(self):
        """
        The method to access the private _eval method.
        Only accessed when the private method is

        appropriately overriden.
        """
        #Sanity check
        if self.cached is None:
            self.cached = self._eval()
```

```
        return self.cached

    def partial_derivative(self, wrt, previous_grad):
        """
        The method to access the private _partial

        derivative method.
        Only accessed when the private method is

        appropriately overriden.
        """
        with add_context(self.name + " PD" + " wrt " + str(wrt)):
            return self._partial_derivative(wrt, previous_grad)
```

Magic methods like add, sub , Mul etc are used to create a feel for the end user similar to that of Numpy.
The *Variable* class is the first and most important inherited class. It is starter of a computational graph, which means there are no children, no passed on values for derivatives and evaluated value is nothing but the value of the variable itself.
Next , operation classes are defined which are used to carry out all the basic mathematical operations on variables.The basic formulation is same, i.e initialize any new attributes that maybe required. Define the eval private method according to the implementation(add two variables for Add class and so on..) and define the partial derivative private method also according to the implementation (e.g for z=x*y dzdx=y).One class definition is given below:

```
class Pow(Node):
    """
    Operation which does power of one node with other
    """
    def __init__(self, first, second, name="Pow"):
        super().__init__([first, second], name)
        self.first = self.children[0]
        self.second = self.children[1]
        self.shape = shape_from_elems(*self.children)

    def _eval(self):
        return np.power(self.first(), self.second())
```

```
    def _partial_derivative(self, wrt, previous_grad):
        if self.first == self.second == wrt:
            return previous_grad * self * (Log(self.first) + 1)
        elif self.first == wrt:
            return previous_grad * self.second *
            Pow(self.first, self.second - 1)
        elif self.second == wrt:
            return previous_grad * Log(self.first) * self
        return 0
```

The operations defined are arithmetic , trigonometric, inverse trigonometric , log , sigmoid, negate, absolute etc. This is in no way as extensive as other commercial packages but it is just efficient as proof of concept. Many more operations can be easily extended similarly.
The most interesting is the implementation of Einsum, an operation which does Einstein summation of arrays. This is an efficient way to implement all tensor operations rather than implementing each operation differently. Wrappers like Matrix multiplication, transpose , trace, etc are defined from this Einsum.
Next are the reshape operations which are used to manipulate an array with itself like reshaping , slicing, padding, summing over a dimension, reducing sum to shape . They are implemented in the similar way the operations have been implemented. One implementation is shown below.

```
class Concat(Node):
    """
    Class to concatenate two arrays
    """
    def __init__(self, a, b, axis=0):
        """
        params:
        a,b: input arrays to be concatenated
        axis: Axis along which concatenation should take place
        """
        #Sanctity check
        assert axis >= 0
        super().__init__([a, b], name="Concat")
        self.a, self.b = self.children
        self.shape = list(self.a.shape)
        self.axis = axis
        self.shape[axis] += self.b.shape[axis]
```

```
    def _eval(self):
        a_val = self.a()
        b_val = self.b()
        return np.concatenate((a_val, b_val), axis=self.axis)

    def _partial_derivative(self, wrt, previous_grad):
        previous_grad = Reshape(previous_grad, self.shape)
        split = self.a.shape[self.axis]

        slice_val = [slice(None, None, None) for _ in range(self.axis + 1)]
        if wrt == self.a:
            slice_val[self.axis] = slice(None, split, None)
            return previous_grad[slice_val]
        elif wrt == self.b:

            slice_val[self.axis] = slice(split, None, None)
            return previous_grad[slice_val]
        return 0
```

Grad is the function where the derivative is actually calculated.It takes in a computational graph,nodes wrt derivatives are desired as a list and outputs a computational graph of the corresponding derivatives.It is independent of the type of computational graph used.It is just concerned with topology of the computational graph which it traverses through all the previous gradients collected using the Node IDs and they are stored in a dictionary. Then , a foldl is performed to get the derivatives in the form of add partials method. To get higher order derivatives, we use grad function twice and so on. It is given below.

```
def grad(top_node, wrt_list, previous_grad=None):
    """
    Transforms the computational graph of top_node into a

    list of computational graphs corresponoding to
    partial derivatives of top_node with respect to all

    variables in wrt_list.

    It delegates the actual implementation of partial
```

```
derivatives to nodes in the computational graph and

doesn't care
how they're implemented.
It can be elegantly implemented using foldl.
Essentially, grad is structural transformation that is

a function *only* of the topology of the computational

graph.

:param top_node: node in the graph whose gradient will

be taken with respect to all variables in wrt_list
:param wrt_list: list of objects, instances of Node,

whose gradient we're looking for
:param previous_grad: incoming gradient to top node,
by default np.ones(top_node.shape)
:return: returns a list of gradients corresponding to

variables in wrt_list
"""
assert isinstance(wrt_list, list) or isinstance(wrt_list, tuple)
if previous_grad is None:
    previous_grad = Variable(np.ones(top_node.shape),

    name=add_sum_name(top_node))

dct = collections.defaultdict(lambda: Variable(0))
dct[top_node] += previous_grad  # add the incoming

gradient for the top node

def add_partials(dct, node):
    for child in set(node.children):  # calc. all

    partial derivs w.r.t. each child and add them to

    child's grads
        dct[child] +=
```

```
            node.partial_derivative(wrt=child,

            previous_grad=dct[node])
        return dct

    dct = functools.reduce(add_partials,

    reverse_topo_sort(top_node), dct)  # basically a foldl

    return [dct[wrt] for wrt in wrt_list]
```

Using this framework , a Neural Network is built such that backprop operations can be done directly under the hood.

3.2 The Neural Network

The Neural Network is implemented as an object such that an object(custom data type) of the model is created for a given number of neurons, number of hidden layers,input dimension and output dimension. First a class of layer is created which will create a layer of Neural Network based on input and output dimensions and all these layers are instantiated and assembled in the Neural Network class definition. Setter and getter methods to change weights are also added with sanity check points.

The architecture for the Neural Network is given by [9] and they called each layer a DGM layer.The architecture is similar to LSTM networks and highway networks discussed in the previous section. The architecture is best elucidated visually in the figures 9 and 10

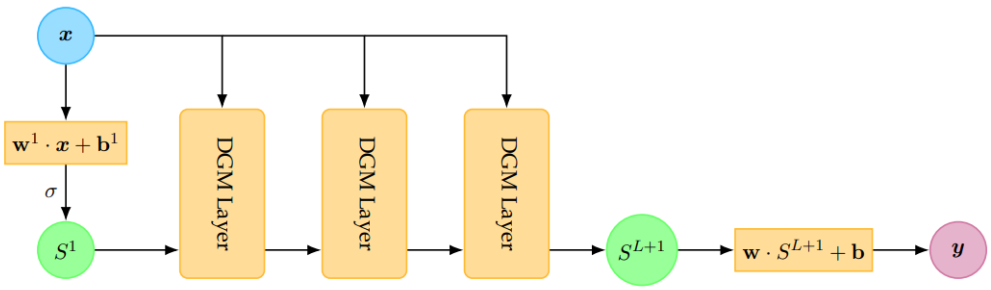


Figure 9: The overall architecture of the Neural Network [2]

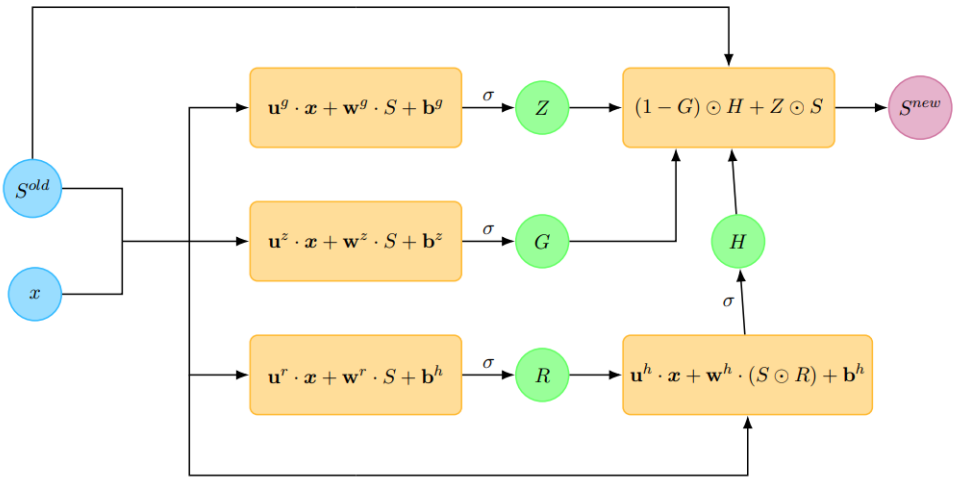


Figure 10: Operations in each layer [2]

Each layer takes in a batch of inputs and output from previous layer (like a Recurrent Neural Network) and they finally culminate into output vector of required dimension. Within each layer the input batch and the output from previous layer is manipulated through a series of operations. u,w,b with varying superscripts form the parameters to be optimized to minimize the loss. Similar to the intuition for LSTMs, each layer produces weights based on the last layer, determining how much of the information gets passed to the next layer. In [9] the authors also argue that including repeated element-wise multiplication of nonlinear functions helps capture "sharp turn" features present in more complicated functions. Note that at every iteration the original input enters into the calculations of every intermediate step, thus decreasing the probability of vanishing gradients of the output function with respect to x [2]. Each layer contains 8 weight matrices and 4 bias vectors when compared to a normal ANN which has only one weight and bias vector gives the intuition that it will be efficient in capturing complex behaviours.The mathematical formulation is given by equation 7

$$
\begin{aligned}
S^1 &= \sigma\left(\mathbf{w}^1 \cdot \boldsymbol{x} + \mathbf{b}^1\right) \\
Z^\ell &= \sigma\left(\mathbf{u}^{z,\ell} \cdot \boldsymbol{x} + \mathbf{w}^{z,\ell} \cdot S^\ell + \mathbf{b}^{z,\ell}\right) && \ell = 1, \ldots, L \\
G^\ell &= \sigma\left(\mathbf{u}^{g,\ell} \cdot \boldsymbol{x} + \mathbf{w}^{g,\ell} \cdot S^\ell + \mathbf{b}^{g,\ell}\right) && \ell = 1, \ldots, L \\
R^\ell &= \sigma\left(\mathbf{u}^{r,\ell} \cdot \boldsymbol{x} + \mathbf{w}^{r,\ell} \cdot S^\ell + \mathbf{b}^{r,\ell}\right) && \ell = 1, \ldots, L \\
H^\ell &= \sigma\left(\mathbf{u}^{h,\ell} \cdot \boldsymbol{x} + \mathbf{w}^{h,\ell} \cdot \left(S^\ell \odot R^\ell\right) + \mathbf{b}^{h,\ell}\right) && \ell = 1, \ldots, L \\
S^{\ell+1} &= \left(1 - G^\ell\right) \odot H^\ell + Z^\ell \odot S^\ell && \ell = 1, \ldots, L \\
f(t, \boldsymbol{x}; \boldsymbol{\theta}) &= \mathbf{w} \cdot S^{L+1} + \mathbf{b}
\end{aligned}
\tag{7}
$$

All the weights are initialized by Xavier initialization which is simply a normal distribution with mean =0 and standard deviation = square root of inverse of mean of input and output dimension. A Neural Network is instantiated as

```
model = NeuralNetLSTM(number of neurons, number of hidden

layers, input dimension,output dimension)
```

3.3 Optimizers

Implementation of optimizers is also done using polymorphism and inheritance. There is one common thing every optimizer used in deep learning does- It takes in parameters and their gradients and does some calculation among them to give a new set of parameters. How these calculations are done so as to increase the efficiency of the optimizer forms the basis for different types of optimizers. Keeping this in mind, a base class *optimizer* is written , which does this basic operation of calling the optimizer with parameters and their gradients as input.

The calculation method(forward pass) is left as virtual method raising a NotImplementedError to be overriden by all the extending classes according to their implementation and those implementation formulas are briefed.The base class is defined as:

```
class optimizer():
    """
    Base class /Parent class for all optimizers
    """
    def _forward_pass(self):
        """
        common polymorphic method for all optimizers.
        "_" denotes that it is private (shouldn't be accessed directly)
        and defining it here with raising error enables it

        to be overriden in child classes.
        """
        raise NotImplementedError
    def __call__(self,params,grad_params):
        """
        This call invokes the private forward class method.
        Input arguments:
        params: The parameters which are being passed to
```

```
        the optimizers as list
        grad_params: The gradients of parameters(in the

        same order) which are being passed to the optimizer as a list
        returns: new parameters after forward pass based

        on optimization algorithm.
        """
        new_params = self._forward_pass(params,grad_params)
        return new_params
```

The equations of optimizers below are taken from [6]

Stochastic Gradient Descent is like a man is walking down a hill, he evaluates which direction is of the steepest descent and takes a step along that direction .Mathematically, the derivative of the loss function is taken and subtracted to the current set of parameters multiplied by learning rate.The update formula is given as $\vartheta = \vartheta \alpha J(\vartheta)$, where J is the loss function, α is the learning rate.
It's advantages include it's simplicity in implementation , understanding and some challenges might be selection of learning rate, same step for all parameters, high variance.

Momentum: When a man is coming down a hill , he gains momentum and his speed increases as he reaches minima,this is precisely the intuition behind Momentum. It accelerates the convergence towards the relevant direction and reduces the fluctuation to the irrelevant direction. One more hyperparameter is used in this method known as momentum(γ). Mathematically , the update rule will be $V(t) = \gamma V(t1) + \alpha . J(\vartheta)$ and $\vartheta = \vartheta V(t)$.
It converges faster than Gradient descent but it adds one more hyperparameter which needs to be manually tuned.

Adagrad: A little intelligent man would like to move more in one direction and less in another direction to move efficiently. This is precisely the intuition behind Adagrad.It changes the learning rate 'η' for each parameter and at every time step 't'.Update formulae are given below.
$g_{t,i} = \nabla_\theta J(\theta_{t,i})$
$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

One of Adagrad's main benefits is that it eliminates the need to manually tune the learning rate. Most implementations use a default value of 0.01 and leave it at that.

Adagrad's main weakness is its accumulation of the squared gradients in the denominator: Since every added term is positive, the accumulated sum keeps growing during training. This in turn causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge[6]. The following algorithms aim to resolve this flaw.

RMSProp : This was developed by Geoff Hinton in an unpublished work but gained significance in deep learning community.RMSprop divides the learning rate by an exponentially decaying average of squared gradients.Learning rate and decaying constant values are usually fixed as .9 and .01. Mathematical formulae are given below

$$\begin{aligned} E[g^2]_t &= 0.9E[g^2]_{t-1} + 0.1g_t^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \end{aligned} \tag{8}$$

Adam: Adam means Adaptive Momentum. A more intelligent man walking down a hill increases his speed while coming down , but he would also slow down near flat surfaces to carefully search for minima. This is the intuition behind Adam.The momentum adapts according to the situation. For this decaying averages of past and past squared gradients are computed as:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \end{aligned} \tag{9}$$

and the update formula: $\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$

The authors propose default values of 0.9 for $\beta_1, 0.999 for \beta_2$.

Adamax: This is purely mathematical update given to the Adam algorithm, instead of using squared norm of past gradients for the vt term ,

infinity norm is used, usually best fit between them is used for more efficiency.The mathematical formulae are given below.

$$\begin{aligned} u_t &= \beta_2^{\infty} v_{t-1} + (1 - \beta_2^{\infty})|g_t|^{\infty} \\ &= \max(\beta_2 \cdot v_{t-1}, |g_t|) \end{aligned} \tag{10}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{u_t}\hat{m}_t \tag{11}$$

3.4 Solving Differential Equations

The approach taken to code the solving of differential equations is almost same for all the differential equations:

1. Instantiation of Neural Net and Optimizer
2. Sampling the points for training
3. Construction of loss function
4. Training the model using the optimizer to lower the loss function
5. Plotting

All the code snippets are taken from the example which solves a simple cantilever beam with concentrated load at the end.

Instantiation of Neural Net and Optimizer: An object of Neural network is created to work with. The depth and number of neurons is selected intuitively based on type of the problem to be solved(it makes sense to take a large NN for complex problems involving higher derivatives, whose solutions are known to be complicated).

```
#Instantiating model and optimizer
model = NeuralNetLSTM(10,1,1,1)
model.set_weights([xavier(i().shape[0],i().shape[1]) for i
   in model.get_weights()])
optimizer= Adamax(len(model.get_weights()))
epochs = 2000
x=sampler(100)
```

Sampling points for training: Random points all along the boundary are needed for training, this next step generates these random points within the limits where the solution is to be found.

```
def sampler(n):
    """
    samples of random data points(uniformly distributed)
    inputs:
    n : number of data points

    returns array of size n

    """
    np.random.seed(0)
    return np.reshape(np.random.uniform(0,10,n),(n,1))
```

Construction of Loss function: Perhaps this is the most important part of code because all the core functionality and concept is coded here.It takes the neural network model and point/s at which it needs to be trained and it penalizes the differential operator of the corresponding operator at the point/s and returns the loss. This can be done either by calculating loss at individual points and returning it or calculating loss at all the points and taking it's mean. Both are implemented. Another classification comes from the possibility of Hard and Soft boundary assignment.
For the same differential equation(soft assignment):
y' = y BCs: y(2) = exp(2) y(3) = exp(3)

The loss function when taking the individual point loss calculation approach is:

```
def loss_domain(model,point):
    """
    Calculates the loss within the domain of the
    differential equation
    inputs:
    model: The Neural Network model to be trained
    point: The point at which loss should be
    calculated(should lie within the domain)
    returns: Squared loss in domain
    """
    point = ad.Variable(np.array([[point]]),name="point")
```

```
    val = model.output(point)
    loss =    ad.grad(val,[point])[0] - val
    #print("loss:",loss())
    return ad.Pow(loss,2)
def loss_boundary(model):
    """
    Calculates loss at the boundaries.
    Inputs:
    model: The Neural Network model to be trained
    returns: Sum of Squared loss at the upper and lower
    boundaries
    """
    #point = ad.Variable(np.array([[0]]),name="point")
    pointu =ad.Variable(np.array([[2]]),name="pointu")
    pointm =ad.Variable(np.array([[3]]),name="pointm")
    #val = model.output(point)-np.array([[1]])
    valu =model.output(pointu)-np.array([[np.exp(2)]])
    valm = model.output(pointm)-np.array([[np.exp(3)]])


    return ad.Pow(valu+valm,2)
```

The loss function when using the mean reduction approach looks like(soft assignment , This method facilitates inclusion of boundary conditions in the same function reducing tracing):

```
def loss_calculator(model,points):
    """
    Calculates the loss within the domain nd boundary of
    the differential equation
    inputs:
    model: The Neural Network model to be trained
    points: The points at which loss should be
    calculated(should lie within the domain)
    returns:Mean Squared loss from all the points in
    domain
    """
    X = ad.Variable(points,"X")

    val = model.output(X)
    f = ad.grad(val,[X])[0] - val
```

```
    lossd = ad.ReduceSumToShape(ad.Pow(f,2),())/100
    fb =model.output(np.array([[2],[3]])) - np.array([[np.exp(2)],[np.exp(3)]])
    lossb = ad.ReduceSumToShape(ad.Pow(fb,2),())
    loss = lossb + lossd
    return loss
```

For the example of cantilever beam with concentrated load at the end the Hard assignment looks like(taking the form such that the boundary conditions are met apriori) :

```
def loss_calculator(model,points):
    """
    Calculates the loss within the domain nd boundary of
    the differential equation
    inputs:
    model: The Neural Network model to be trained
    points: The points at which loss should be
    calculated(should lie within the domain)
    returns:Mean Squared loss from all the points in
    domain [0,10]
    """
    X = ad.Variable(points,"X")

    val = (10-X)*(10-X)*model.output(X)
    #Force (S.I Units)
    p = 10000
    #Flexural Rigidity - EI
    F = 2*0.000005*200000*1000000
    temp = p/F

    f = (diff_n_times(val,X,2)) + ((temp*ad.Pow(X,2)))
    print(f.shape)
    lossd = ad.ReduceSumToShape(ad.Pow(f,2),())/100
    Xb = ad.Variable(np.array([[10]]))
    fb1 = model.output(Xb)
    lossb1 =  ad.ReduceSumToShape(ad.Pow(fb1,2),())
    fb2 = ad.grad(model.output(Xb),[Xb])[0]
    lossb2 =  ad.ReduceSumToShape(ad.Pow(fb2,2),())

    return lossd
```

Solving higher dimensional differential equations is tedious task and it is more or less proceeded in the same way. The sampling should be done in the whole domain ,which will cause construction of loss function more complicated leading to very large computation times. There is another way of solving higher dimensional equations.
This will be explained using the example of Burgers equation(one space and one time dimension). It is given as :

$$\begin{aligned} &u_t + uu_x - (0.01/\pi)u_{xx} = 0, \quad x \in [-1, 1], \quad t \in [0, 1], \\ &u(0, x) = -\sin(\pi x) \\ &u(t, -1) = u(t, 1) = 0 \end{aligned} \tag{12}$$

This equation was proved extremely hard to resolve using conventional Runge-Kutta techniques because of highly non-linear nature and shock layer formation at t=0.4 seconds. This is easy to code using Neural Networks but it requires thousands of points and very deep NN which will take hours of computational time and it will inevitably break the framework(since it is very basic one). Hence a clever exploitation is done utilizing the best of the two worlds: The equation is discretized in one dimension(time dimension because we usually need continuous solution in space at discrete time steps) using conventional Runge-Kutta stepping schemes and a Neural Network is approximated upon this discretized equation starting from initial time step. The discretized equation will look like [8]:

$$\mathcal{N}\left[u^{n+c_j}\right] = u^{n+c_j}u_x^{n+c_j} - (0.01/\pi)u_{xx}^{n+c_j} \tag{13}$$

where

$$\begin{aligned} &u^{n+c_i} = u^n - \Delta t \sum_{j=1}^{q} a_{ij}\mathcal{N}\left[u^{n+c_j}\right], \quad i = 1, \ldots, q \\ &u^{n+1} = u^n - \Delta t \sum_{j=1}^{q} b_j\mathcal{N}\left[u^{n+c_j}\right] \end{aligned} \tag{14}$$

and,

$$u^{n+c_j}(x) = u\left(t^n + c_j\Delta t, x\right) \text{ for } j = 1, \ldots, q \tag{15}$$

A stepping scheme with q=32 is selected and initial data at t=0.1s is taken from [8]

The loss function will be :

```
def loss_calculator(model,x0,u0,IRK_weights,):
    """
```

```
calculates squared loss at all data points in both
domain and boundary .
Inputs:
model: The model to be trained
x0   : Data points
u0   : Solution at 0.1 second
IRK Weights : Butcher Tableau of corresponding Runge-
Kutta weight matrix
returns: mean squared loss
"""
#converting data points into suitable type
X=ad.Variable(x0,name="X")
U = model.output(X)
U1 = U[:,:-1]
#Instantiating dummy variables to get forward
gradients
dummy1 = ad.Variable(np.ones((100,32)),name="dummy1")
dummy2 =ad.Variable(np.ones((100,33)),name="dummy2")
dummy3= np.ones((100,32))
#g = ad.grad(U,[X],previous_grad=dummy2)[0]
#print("g:",g().shape)
gx = ad.grad(ad.grad(U,[X],dummy2)[0],[dummy2])[0]
ux = gx[:,:-1]
#print("ux",ux().shape)
#g1 = ad.grad(g,[X],previous_grad=dummy1)[0]
#print("g1",g1().shape)
gxx= ad.grad(ad.grad(gx,[X],dummy2)[0],[dummy2])[0]
uxx = gxx[:,:-1]

#print("uxx",uxx().shape)
#F = -U1*g + (0.01/np.pi)*g1
F = -U1*ux + ((0.01/np.pi)*uxx)
#Formulate the loss
temp =0.4*ad.MatMul(F,IRK_weights.transpose())
U0 = U - temp
u0 = ad.Variable(u0,name="u0")
#val = ad.ReduceSumToShape(U0,(250,1))
#squared Sum over all axes
lossd = ad.ReduceSumToShape(ad.Pow(U0 - u0,2),(1,1))
#print(lossd())
X1 = ad.Variable(np.vstack((-1.0,+1.0)),name="X1")
```

```
    Ub = model.output(X1)
    #Loss at boundary squared sum over all axes
    lossb = ad.ReduceSumToShape(ad.Pow(Ub,2),(1,1))
    loss = lossd + lossb



    return loss
```

Allen-Cahn which is used in solubility of phases in metals is also solved using the similar discretization technique and it involves third order derivatives and boundary conditions also involve derivatives it is given as:

$$\begin{aligned} &u_t - 0.0001u_{xx} + 5u^3 - 5u = 0, \quad x \in [-1,1], \quad t \in [0,1], \\ &u(0,x) = x^2\cos(\pi x) \\ &u(t,-1) = u(t,1) \\ &u_x(t,-1) = u_x(t,1) \end{aligned} \tag{16}$$

and time discretized loss function is :

```
def loss_calculator(model,x0,u0,IRK_weights):
    """
    calculates squared loss at all data points in both
    domain and boundary .
    Inputs:
    model: The model to be trained
    x0   : Data points
    u0   : Solution at 0.1 second
    IRK Weights : Butcher Tableau of corresponding Runge-
    Kutta weight matrix
    returns: mean squared loss
    """
    #converting data points into suitable type
    X=ad.Variable(x0,name="X")
    U = model.output(X)
    U1 = U[:,:-1]
    #Instantiating dummy variables to enable forward
    gardients
    dummy1 = ad.Variable(np.ones((150,32)),name="dummy1")
    dummy2 =ad.Variable(np.ones((150,33)),name="dummy2")
    dummy3 = ad.Variable(np.ones((2,33)),name="dummy3")
    #Taking gradients and formulating the loss function
```

```
gx = ad.grad(ad.grad(U,[X],dummy2)[0],[dummy2])[0]
ux = gx[:,:-1]
gxx= ad.grad(ad.grad(gx,[X],dummy2)[0],[dummy2])[0]
uxx = gxx[:,:-1]
F = 5.0*U1 - 5.0*ad.Pow(U1,3) + 0.0001*uxx
temp =0.4*ad.MatMul(F,IRK_weights.transpose())
#print(temp.shape)
U0 = U - temp
u0 = ad.Variable(u0,name="u0")
#loss vector on domain
vald = u0-U0
#summing over all the axes
lossd = ad.ReduceSumToShape(ad.Pow(vald,2),())

X1 = ad.Variable(np.vstack((-1.0,+1.0)),name="X1")
Ub = model.output(X1)

ubx = ad.grad(ad.grad(Ub,[X1],dummy3)[0],[dummy3])[0]

#Loss vector on boundary
loss_b_val = Ub[0,:] - Ub[1,:]
#print(loss_b_val.shape)
lossb = ad.ReduceSumToShape(ad.Pow(loss_b_val,2),())
#print(lossb())
#Loss vector on another boundary
lossbx_val = ubx[0,:] - ubx[1,:]
lossbx = ad.ReduceSumToShape(ad.Pow(lossbx_val,2),())
#print(lossbx())
return lossb + lossbx + lossd
```

Next step is training the Neural Network by using optimizers. Gradients are required by the optimizers , they are calculated using the *grad* function. Derivatives are taken with respect to the weights (showing that backprop is just a special case for autodiff) and training is done until a termination criterion is met.The plotting involves saving the output from trained NN into an array and plotting it against the given inputs(scatter plot for random inputs).

4 Testing and Validation

Testing is done to make sure every functionality of code is working as expected without any discrepancies by making sure that it gives the expected output for a known set of inputs in various scenarios . Each functionality is tested thoroughly using the *PyTest* environment.

4.1 Testing the Autodiff Framework

The Autodiff framework has mathematical operations and array manipulators. Since every operation is implemented as a class, every class is tested by instantiating its object. basically every test case checks three things:

1. Is the node created of expected type?
2. Is the value evaluated at the node is correct against known set of inputs?
3. Is the gradient being evaluated at that node is as expected?

Therefore every test case is asserted for these three values to be equal. One such test case is:

```
def test_Mul1_irrational():
    """
    Aim:Test the Mul operation (if the value is correct
    and derivatives are passed as expected)
    Expected:pi ,dzdx=pi,dzdy=1
    Obtained:pi,dzdx=pi,dzdy=1
    Remark:multiplying an irrational integer to check
    robustness of kernel
    """
    x = 1
    y = np.pi
    Z = x*y
    X = Variable(x,"X")
    Y = Variable(y,"Y")
    Z1 = Mul(X,Y)
    dzdx,dzdy = grad(Z1,[X,Y])
    assert isinstance(Z1,Mul) and Z1()==np.pi and
    dzdx()==y and dzdy()==x
```

Every operation is tested for the above checkpoints for a multitude of scenarios (rational inputs, irrational inputs, higher dimensional arrays etc) to ensure the robustness and sanctity of the performance of the code.
An important note here is to be given about the handling of infinities. What I found was that the derivatives are discontinuous even for the simple functions and they are to occur more often than not even in a simple Neural Network problem and it will be very inconvenient for the program to throw infinities and manually checking them all. To circumvent this I used a similar approach used in Tensorflow, don't take the value of evaluation or/derivative of the function at discontinuity but shift the whole curve such that the value is evaluated near small neighbourhood of the discontinuity effectively eradicating unwanted infinities. This ofcourse decreases accuracy since numerical kernel now evaluates a shifted value, but this error is very less (order of eighth digit after decimal) and this is a good trade-off . Hence in some cases the assertion done for the difference between expected value and obtained value to be less than some tolerance (generally 1e-8).

One typical test case in this scenario is:

```
def test_recipr_scalar():
    """
    Aim: To test the operation Reciprocal(which basically
    does element-wise reciprocal)
    Obtained: 0.19999999999996 and dy = -0.04
    Expected: 0.2 and dy = -0.04
    Remarks: This small deviation marks the distinctive
    feature employed in all the operations which might
    have to deal with unwanted infinities. In this case
    division by zero is avoided by adding a very small
    innate error of 1e-12
    """
    x= 5
    X = Variable(x,"X")
    Y= Recipr(X)
    dy = grad(Y,[X])[0]



    assert isinstance(Y,Recipr) and np.abs(Y() - (1/
    (5+1e-12))) < 0.0000001 and np.abs(dy()  -
    np.array(-0.04)) < 0.000000001
```

Every typical case where the function theoretically breaks but is expected to evade it according to implementation is tested for every function to ensure robustness.

4.2 Testing Neural Network Architecture

The developed Neural Network architecture is tested by taking a Neural Network and changing it's weights to a known set and asserting against expected output from this known set of inputs and weights. This is pretty simple and all the test cases do the same thing

4.3 Testing Optimizers

Basically optimizer is an object which performs a step calculation and returns parameters. So, It is tested for a step. For a known input values of gradients and parameters , does the optimizer take step as expected? This has been tested for different parameters and gradients (integers and matrices) to ensure correct behaviour. For optimizers whose internal parameters change for every iteration , the parameters are also asserted. For Adamax the same principle to evade infinities from Autodiff is used and hence absolute difference is asserted to be less than tolerance. An example test case:

```
def test_Adam_NN():
    """
    test Adam optimizer with learning rate 0.1 , beta1(The
    exponential decay rate for the first moment estimates)
    = 0.9, beta2(The exponential decay rate for the
    second-moment estimates ) =0.999
    tested for : randomly initialized weight matrices(x)
    of a Neural Network with number of neurons:5 , number
    of layers:1, input dimension: 5, output dimension:6
    and gradients(dx) equal to ones of same shape
                and momentum and velocity vectors which
                are unique prospects of Adam.
    tested with : Hand calculated values based on
    formulae:
                  new momentum = beta1*momentum + (1-
                  beta1)*gradients
                  new velocity = beta2*velocity + (1-
                  beta2)*gradients*gradients
                  accumulation = learning rate * sqrt(1-
```

```
                beta2**iteration)/(1-beta1**iteration)
                new parameters = parameters -
                accumulation*new momentum/sqrt(new
                velocity)
"""
#instantiating NN
model = NeuralNetLSTM(5,0,5,6)
x = model.get_weights()
#instantiating optimizer
opt = Adam(len(x),lr=0.1)
dx=[]
for i in range(len(x)):
    dx.append(np.ones_like(x[i]))
nx=[]
for i in range(len(x)):
    #gradients of same shape and value 1
    temp = x[i] - 0.03162277343940645*(0.1/
    np.sqrt((0.001+1e-8)))*dx[i]
    nx.append(temp())
#Calling optimizer
nx_temp = opt(x,dx)
nx_opt = [i() for i in nx_temp]
print(nx[0])
print(nx_opt[0])
m = [0.1*i for i in dx]
v = [0.001*i for i in dx]
for i in range(len(x)):
    assert np.all(np.less(np.abs(m[i]-
    opt.momentum[i]),0.000000000001)) and
    np.all(np.less(np.abs(v[i]-opt.velocity[i]),
    0.000000000001)) \
        and np.all(np.less(np.abs(nx[i]-nx_opt[i]),
        1e-6))
```

4.4 Validation Cases

Tests made sure that the functionalities of code are working as intended, validation cases are carried out to show that the method of solving the differential equations works. Cases are built to solve very small problems with

easily verifiable solution to show that the proposed method actually works. First , functional approximation theorem should be validated which is the basis of solving differential equations using Neural Networks. Basically we need to show that any arbitrary function can be approximated by a deep enough Neural Network, this has been done for a sine function and parabola. The results are shown in the figures 11 and 12 :

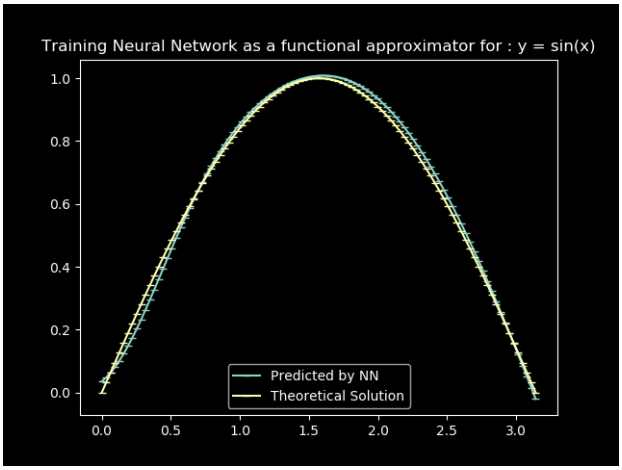


Figure 11: Neural network as functional approximator for a Sine function

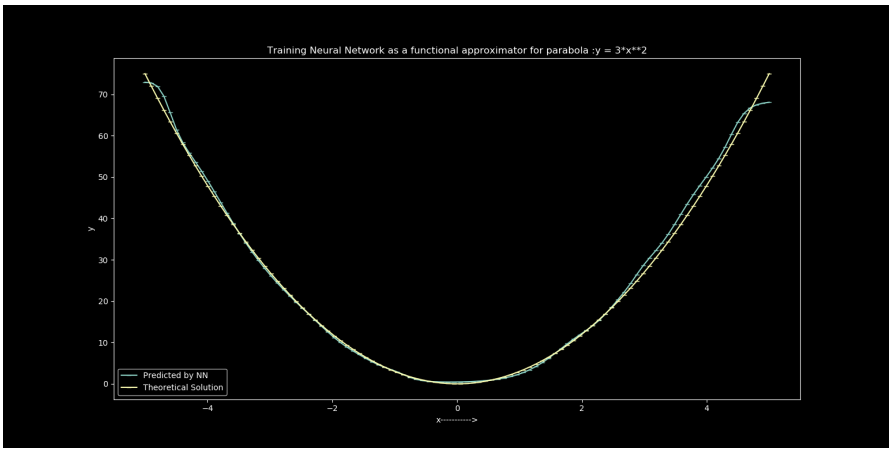


Figure 12: Neural network as functional approximator for a parabola function

The optimizers are validated by using them for optimizing the value of a well known Rosenbrock function and also for optimizing a general linear parameter space function(sinx + cosx + x) the results are shown in figures

13 and 14. This shows that infact all the optimizers work correctly but some characteristics about their performance can also be deduced from the results, which will be discussed later.

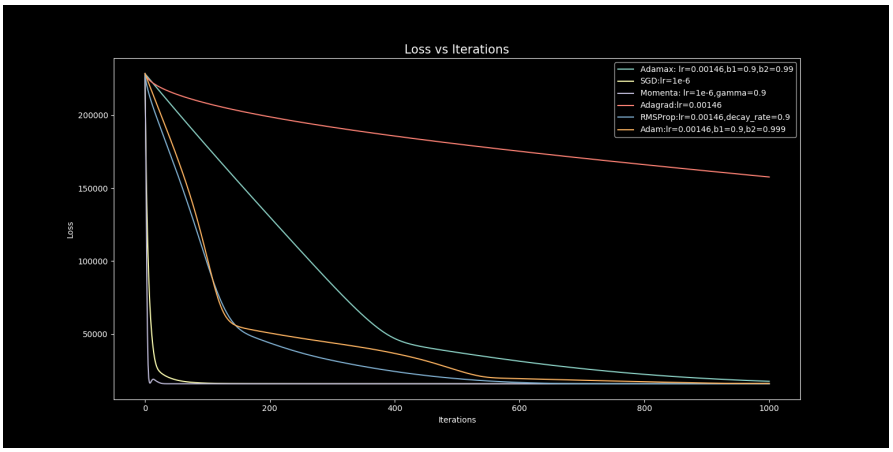


Figure 13: Training a model using all optimizers for sin(x)+cos(x)+x from same starting point

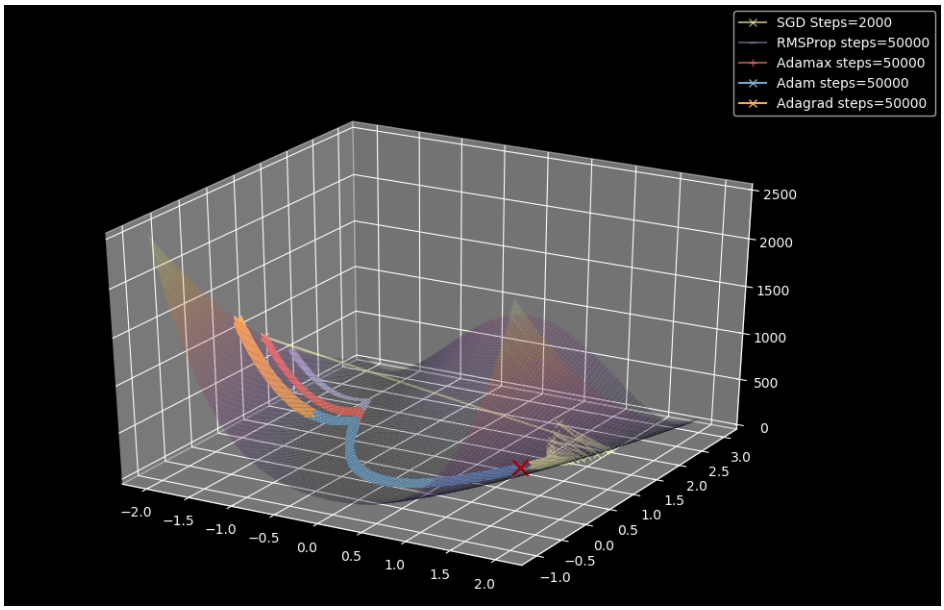


Figure 14: All optimizers minimizing the Rosenbrock function from different starting points

5 Results and Discussion

5.1 Results

The differential equations which are solved :

1. y' = 6x with constant c =0.
   Strategy : Training at Individual points, No Boundary conditions. The result is given in 12

2. y' = y with Boundary conditions y(2)=exp(2),y(3)=exp(3).
   Strategy : Training at individual data points, Soft Assignment of BCs . The result is given in 15

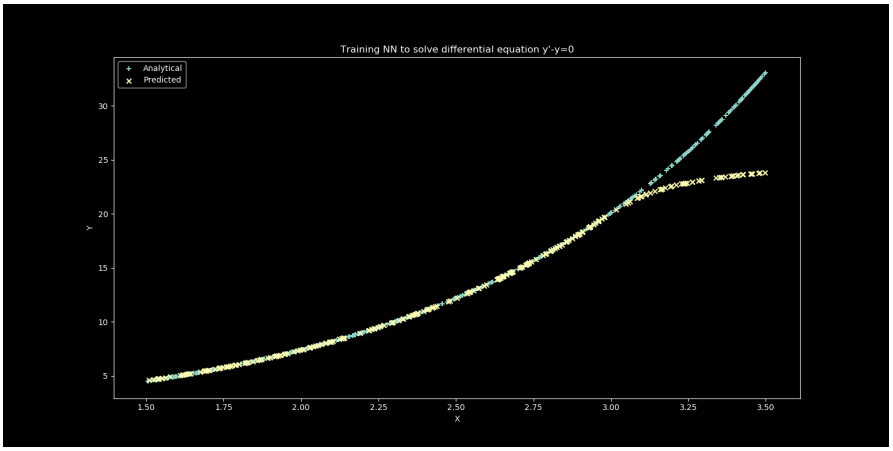


Figure 15: Solving differential equation

3. y' = y with Boundary conditions y(2)=exp(2),y(3)=exp(3).
   strategy : Mean reduction of loss from all points,Soft assignment of BCs. The result is given in 16.

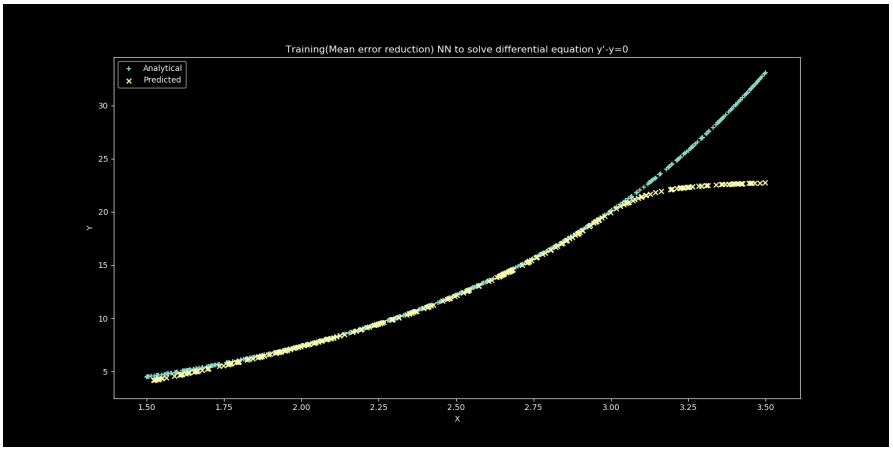


Figure 16: solving differential equation

4. y"-y = 0 with BCs y(0)=0 and y(1)=1 Strategy: Training at individual points, Hard assignment of BCs. The result is given in 17

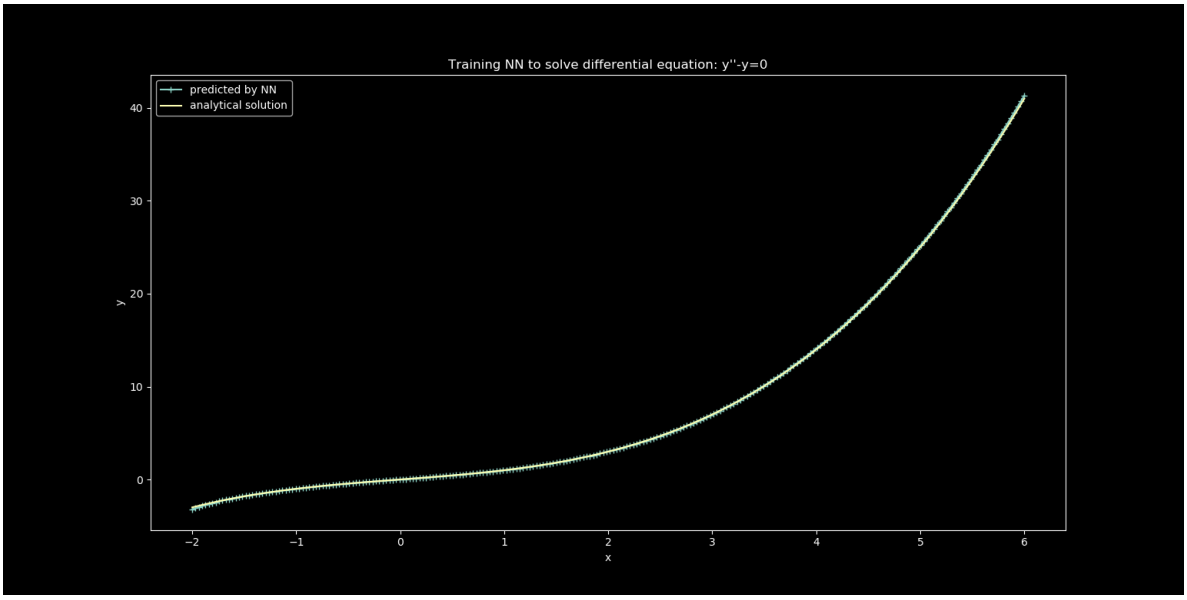


Figure 17: Higher order differential equation

5. Cantilever Beam with uniformly distributed load with parameters: x= 10m, E(Youngs Modulus) = 200000MPa, I(Moment of Inertia in transverse direction) = 0.000005 ,
UDL = 10KN/m .

strategy: Mean reduction of loss from all points,Hard assignment of BCs. The result is given in 18

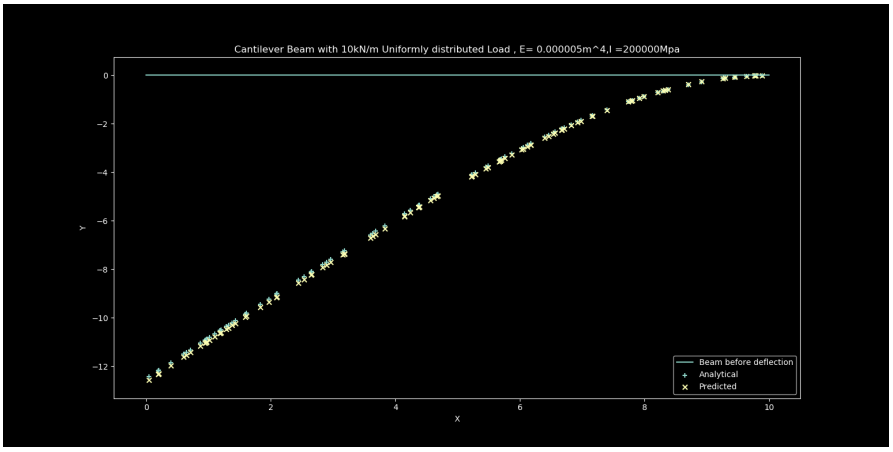


Figure 18: Cantilever Beam with UDL

6. Cantilever Beam with point load at end with parameters : E(Youngs Modulus) = 200000MPa, I(Monent of Inertia in transverse direction) = 0.000005.
, p(Point load) = 10KN
Strategy: Mean reduction of loss from all points,Hard assignment of BCs. The result is given in 19

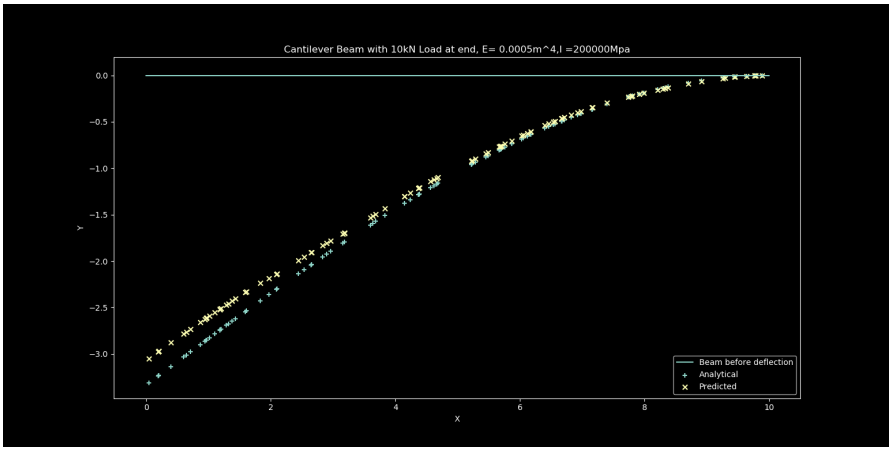


Figure 19: Cantilever Beam with load on end

7. Simply supported beam with uniformly distributed load with parame-

ters:x= 10m, E(Youngs Modulus) = 200000MPa, I(Monent of Inertia in transverse direction) = 0.000005 m.
p(Point load) = 10KN/m Strategy: Mean reduction of loss from all points,Hard assignment of BCs. The result is given 20

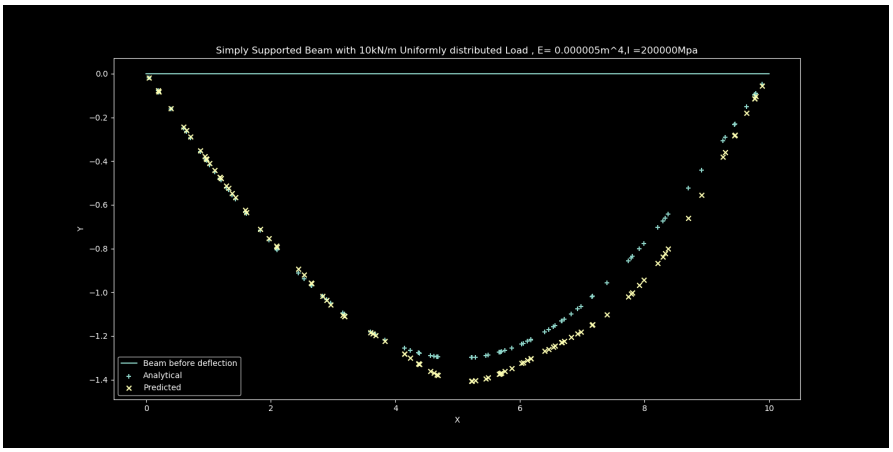


Figure 20: Simply Supported Beam

8. Burgers Equation:

$$\begin{aligned} &u_t + uu_x - (0.01/\pi)u_{xx} = 0, \quad x \in [-1, 1], \quad t \in [0, 1], \\ &u(0, x) = -\sin(\pi x) \\ &u(t, -1) = u(t, 1) = 0 \end{aligned} \tag{17}$$

Strategy: Time discretized , RK-32 stepping scheme ,Mean Reduction of error, Soft assignment of BCs. The result is given in 21. Application: Erosion of Material from solid surface, Acoustic properties of material , Fluid Mechanics etc.

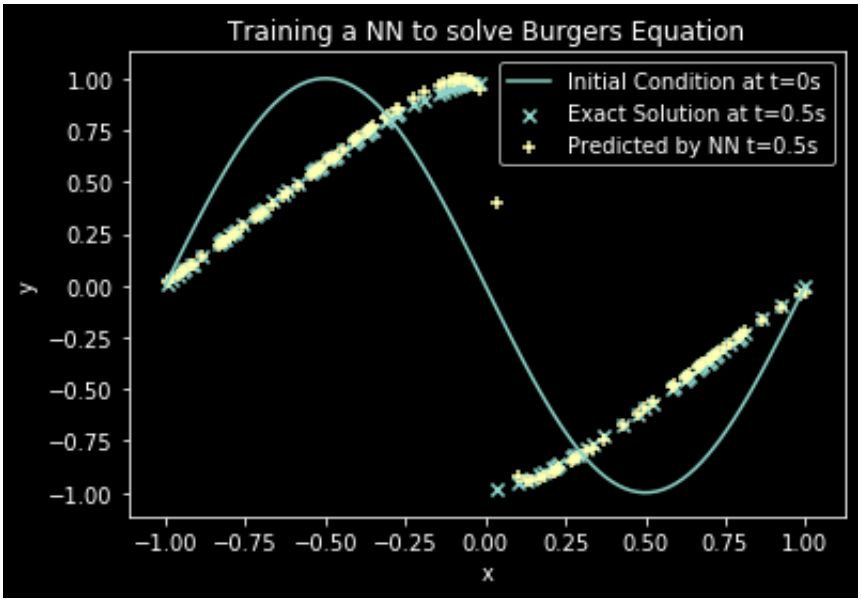


Figure 21: Solving Burgers Equation

9. Allen Cahn-Equation:

$$\begin{aligned} &u_t - 0.0001u_{xx} + 5u^3 - 5u = 0, \quad x \in [-1, 1], \quad t \in [0, 1], \\ &u(0, x) = x^2 \cos(\pi x) \\ &u(t, -1) = u(t, 1) \\ &u_x(t, -1) = u_x(t, 1) \end{aligned} \tag{18}$$

Strategy: Time discretized , RK-32 stepping scheme ,Mean Reduction of error, Soft assignment of BCs. The obtained result after 5000 Iterations is shown in 22. Application:phase separation in multi-component alloy systems

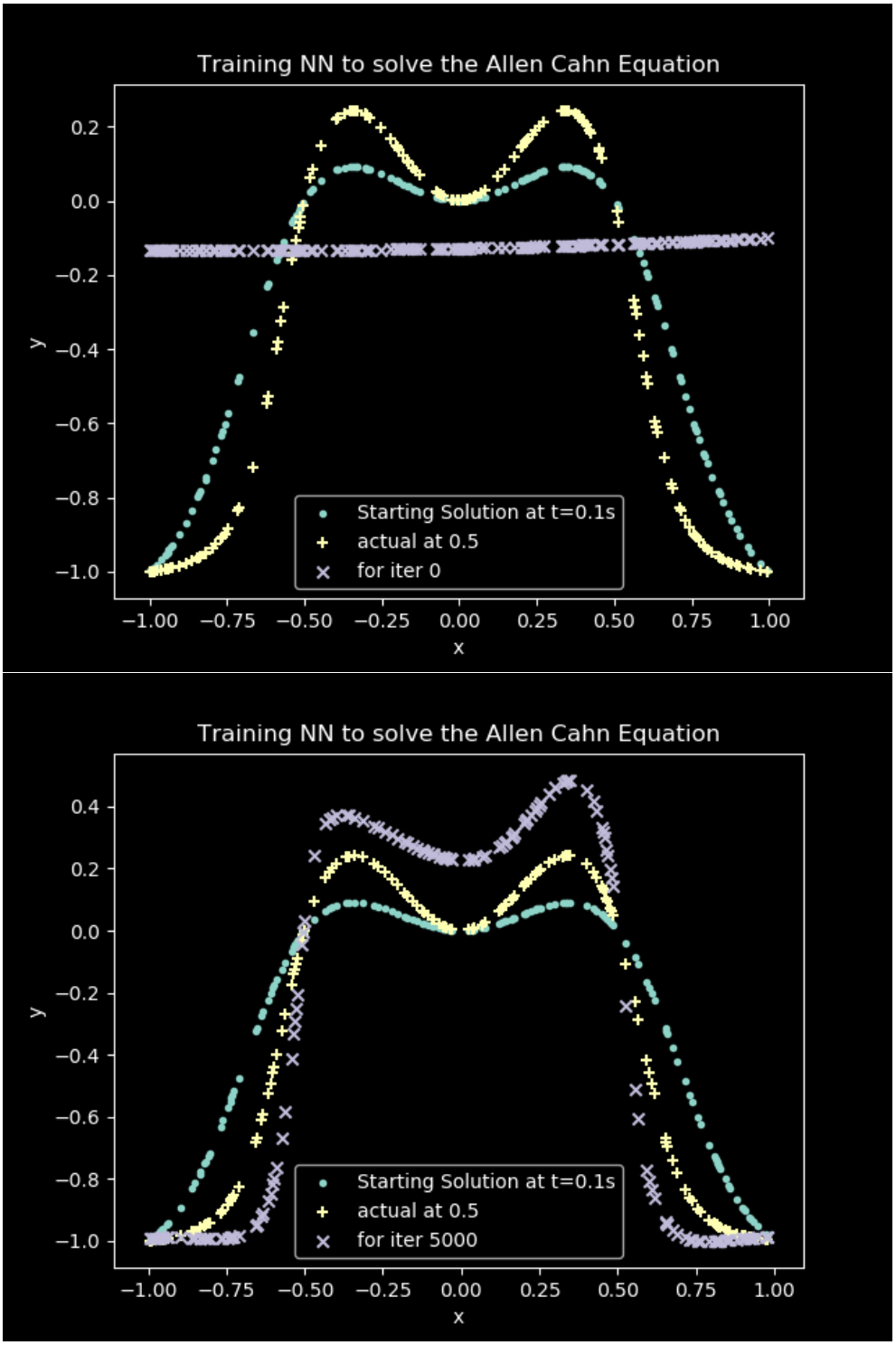


Figure 22: Solving Allen-Cahn Equation

5.2 Inferences

1. There is no prescribed rule for the size of the NN and there is no overfitting problem because as the loss decreases , the NN reaches more and more to the solution of the differential equation. A small NN won't be sufficient to capture a complex behaviour and a large NN will be redundant for a simple contour. Hence the size of NN should be appropriately selected.

2. Hard assignment of BCs by definition gives more accurate answer than soft assignment since in soft assignment the BCs are only approximately satisfied. But, Hard assignment is difficult to formulate and sometimes not available for complicated problems and soft assignment can be formulated for almost any type of problem.

3. The figures 15 and 16 show an interesting inference. The solution is plotted even for a region way outside of the training region and the deviations are clearly identified. The solution is only applicable for the region in which the solution is trained and to be more precise , the sampling strategy matters and even within the trained region , sampling using uniform distribution gives satisfactory results more than say, log-normal distribution of sampling points.

   Time Analysis: The grad function traverses along the computational graph sequentially for every point in every iteration.Hence it is very time taking process. The programs of Burgers equation and Allen-Cahn equation take hours and hours to complete 5000 iterations. Hence it is intuitive to assume bottle necking occurs during gradient calculation.Time analysis on two differential equations given in figures 16 and 15 using the *Snakeviz* visualization tool, which visualizes the log file generated by *cProfile*. The same differential equation is solved using two different strategies(descent step at every point vs Mean-reduction of loss) to check which process is more efficient computationally. While there is no change in accuracy , the time taken by mean reduction is considerably less than taking descent step at every point. This is shown in the figures 23 and 24 .Clear bottle necking is observed during calling of the gradient function.

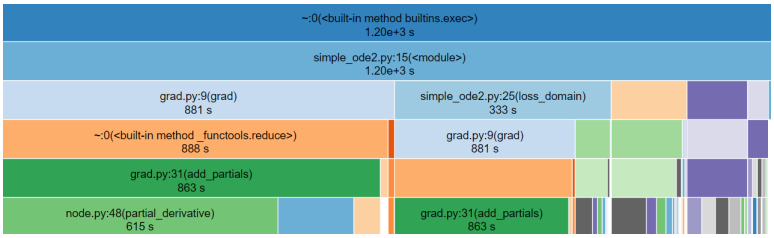
4. 

Figure 23: Times taken by individual functions when the equation is solved by taking single descent step at every point

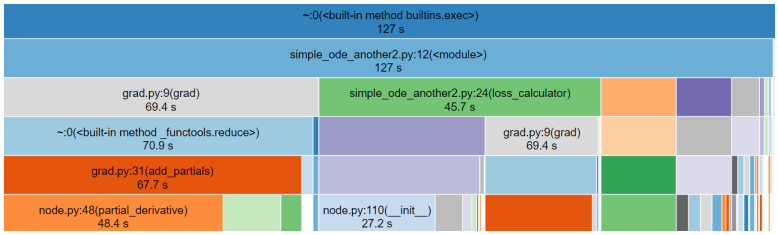


Figure 24: Times taken by individual functions when the same equation is solved by taking mean reduction in error.

5. Selection of optimizer is also totally dependent on the parameter space in which the optimization takes place. SGD,Momentum are very good for linear convex spaces and are very erratic for non-linear ones. Adagrad is good conceptually but it has a big flaw. Adagrad's main weakness is its accumulation of the squared gradients in the denominator: Since every added term is positive, the accumulated sum keeps growing during training. This in turn causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge. [6]. Adam , Adamax and RMSProp solve this problem and behave more or less in the same way , shown in the figure 14. Generally, Adam is preferred when going in unknown terrain.

6. Discretization of differential equation in one dimension and taking steps using NN shows very interesting implications. The problem with classical techniques is the time-steps are usually confined to be small due to stability constraints for explicit schemes or computational complexity constrains for implicit formulations.These constraints become more severe as the total number of Runge–Kutta stages q is increased, and, for most problems of practical interest, one needs to take thousands

to millions of such steps until the solution is resolved up to a desired final time. Using only normal NN to solve requires thousands of collocation points and therefore high compuational effort. Discretized version of solving differential equation evades these two problems and brings together the best of both approaches.In sharp contrast to classical methods, here we can employ implicit Runge–Kutta schemes with an arbitrarily large number of stages at effectively very little extra cost. This enables us to take very large time steps while retaining stability and high predictive accuracy, therefore allowing us to resolve the entire spatio-temporal solution in a single step [8].

7. Simple first order differential equations take minutes to decrease the loss to a satisfying minima. Second order equations(figure 21) , even with time - discretized version take hours to converge to a satisfactory solution, hence the termination is set to a fixed number of iterations(here 5000). Even for 5000 iterations , third order equations (figure 22) couldn't give satisfactory results as a clear discrepancy between analytical and obtained solution can be seen. The solution is converging towards exact solution, but it takes hours and hours. The convergence to analytical solution is seen by comparing the solution at iteration 0 and iteration 5000. This is to show that , this framework is to be taken as proof of concept and this can be used for solving actual materials science related problems efficiently by extending it's multi-processing and GPU capabilities.

6 Promised milestones during proposal

In retrospect,almost everything which was promised during the proposal has been done. These include implementation of Automatic Differentiation, LSTM Neural Network, Optimizers and solving different types of differential equations and one Burgers equation.

I proposed one method to solve the differential equations but here I thought it would be more meaningful to implement and compare different methods like hard assignment , soft assignment as well as single step , mean reduction and time-discretized methods also. 6 optimizers have been implemented(2 were proposed). Testing was done accurately as proposed.

I did not solve Schrödinger equation as proposed because I thought Allen-Cahn equation is relevant to materials sciences and it is more complicated than Schrödinger equation.(The Allen-Cahn equation contains third order derivatives and boundary conditions also include spatial derivatives, in contrast the Schrödinger equation is a second order equation and second order equation was already solved in the form of Burgers equation.)

7 Conclusion and Future scope

This method provides an interesting perspective in solving differential equations. Using Neural networks without any training data is a new paradigm in deep learning. It has produced promising results. But, this can never be an alternative to classical finite element, finite volume methods. These methods have been perfected over last half century. The real interesting thing is that this method is easily formulated and it can co-exist with the aforementioned classical methods, as illustrated in solving the Burgers and Allen-Cahn equations. This co-existence can be further developed due to rapid increase in Machine/Deep learning and such models can be embedded into commercial FEM/FVM tools. [8]

The Autodiff framework implemented here is very primitive and can be easily broken by complicated problems. Modern packages like Tensorflow and PyTorch are extending this core concept using multi-processing,multi-threading and GPU processing techniques providing very elegant and fast calculations.This is just like a very basic version of Tensorflow stripped out of it's high-performance and high-level operations.

There are still many unanswered questions like what is the effect of local-minima, different activation functions,loss formulation?. Is it possible to solve highly oscillating PDEs?. More work is needed to answer these questions.

I hope this work opens a new paradigm of deep learning without data and instead using behaviour of the models(physics of the models) and it's co-existence with classical techniques for Computational material scientists.

8 Manual

This project has :

1. The Automatic Differentiation Package(AutoDiff)
2. The Neural Network Architecture(created using AutoDiff)
3. Optimizers used to train the Neural Network.
4. The solved problems in a few different implementations.

How to install the Package?

1. Navigate to the folder containing the "setup.py" file
2. Execute the command

```
pip install .
```

How to run the test cases?

1. Navigate to the folder containing the files of test cases(named with suffix tests)
2. Execute pytest filename.py , for example,

```
pytest tests_ops.py
```

External Packages Required apriori: Numpy and Matplotlib and sys .

Remaining all the files can be run normally provided the package is installed along with numpy and matlplotlib, since they don't require any user input(If given as user input, loose-ends in computational graphs might occur.) .

References

[1] Chollet Francois. *Deep learning with Python.* Manning Publications Company, 2017/12.

[2] A. Al-Aradi, Adolfo Correia, D. Naiff, G. Jardim, and Yuri F. Saporito. Solving nonlinear and high-dimensional partial differential equations via deep learning. *arXiv: Computational Finance*, 2018.

[3] https://paulvanderlaken.com/2017/10/16/neural-networks-101/.

[4] Shruti Jadon. Introduction to different activation functions for deep learning. *Medium, Augmenting Humanity*, 16, 2018.

[5] Cambridge Coding Academy. Deep learing for complete beginners: Recognising handwritten digits.

[6] https://towardsdatascience.com/optimizers-for-training-neural-network 59450d71caf6. Various optimization algorithms for training neural network.

[7] I. Lagaris, A. Likas, and D. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE transactions on neural networks*, 9 5:987–1000, 1998.

[8] M. Raissi, P. Perdikaris, and G. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *J. Comput. Phys.*, 378:686–707, 2019.

[9] Justin Sirignano and Konstantinos Spiliopoulos. Dgm: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375:1339–1364, 2018.

[10] G Cybenkot. Approximation by superpositions of a sigmoidal function *. 2006.

[11] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Radul, and J. Siskind. Automatic differentiation in machine learning: a survey. *J. Mach. Learn. Res.*, 18:153:1–153:43, 2017.