**ON SOLVING DIFFERENTIAL EQUATIONS WITH NEURAL NETWORKS**

**On solving differential equations with Neural Networks**

CS584 Final Report

Edvin Beqari and Doug Williamson

GMU  December 7, 2020

**Introduction**

The scientific and engineering world is driven by formulas which attempt to describe the physical universe we live in and interact with on a daily basis. There are static formulas which describe steady state events or properties, as well as dynamic formulas which describe how properties change over time or location, as well as other factors. Dynamic formulas include terms which describe how each parameter changes in accordance with the change in another parameter and are represented as differential equations. For example, the motion of a pendulum must balance the force of gravity against the angular momentum of the pendulum, both of which vary based on the position of the pendulum within its arc of motion.

Some differential equation can be solved analytically using separation of variables, or with numerical domain discretization schemas. However, as the number of input variables increases, manual solutions become correspondingly harder to solve. In such cases neural networks and machine learning strategies have proven to overcome many problems encountered by standard methods.

**Automatic Differentiation**

Complex equations can be very difficult to solve and differential equations are no exception. For example, modeling flight trajectories and behavior of aircraft requires a series of partial differential equations to handle six degrees of freedom as well as the external forces acting on the aircraft such as wind speed and drag. While it is possible to program each derivative equation into the computer, it is far easier to use a program which calculates all the gradients. An encoder allows the user to enter a formula and then system breaks the formula down into components using the chain rule from calculus. These components are then converted into differential equations via a series of rules, the end result of which is a fully encoded graph network to execute. In order to understand how the auto differentiation process works, a couple of concepts need to be explained. This explanation uses an example presented by Rufflewind1.

Given the following formula

|  |
| --- |
|  |

The first step is to apply basic algebra and break down complex equations into components that the computer recognizes. Following the example, the formula breaks down as follows:

|  |
| --- |
|  |

where  and  are our input variables and is the graph output. Each of these components are easily handled by the computer program. No matter how complex the initial equation, it can always be decomposed into simple components.

The second important element is the set of rules which are used to convert our formula into its differential form. More specifically, the rules convert the components of our formula into their differential forms. The rules are derived from the chain rule from calculus and are as follows:

|  |
| --- |
|  |
|  |

Note that  is shorthand for  where  can represent any arbitrary variable. Using these rules, the auto encoder can convert the components of our initial formula into differential form.

|  |
| --- |
|  |

The outputs of this process support Forward Mode auto differentiation. As the name implies, calculations are performed sequentially from the inputs to the outputs. Forward Mode requires one pass through the network for each input. Reverse mode inverts the problem, working from the output back to the inputs. Calculus allows us to invert our formulas (i.e., exchange numerators and denominators), so the principle is straightforward, but the mathematics are too complicated to present here. Fortunately, Keras already provides libraries for performing Reverse Mode Auto Differentiation, so it is not necessary to program it ourselves. Reverse Mode generates all the input derivatives in a single pass. Accordingly, this process is most effective when there are many inputs and few (or one) outputs. Since many scientific problems take multiple inputs to produce a single output value, this has wide applicability.

**The Math**

This section introduces the problem of solving differential equations as a form of “self-supervised’ learning. The report assumes familiarity with Feed Forward Neural Network and Differential Equations. This section covers two approaches. A naïve approach where the cost function is simply the mean sum of the differential operator, its boundary and initial conditions, and a classical cost in the cases where there are real observations. The second approach was introduced first by Lagaris, Likas, and Fotiadis2 where a two-part trial solution is introduced where one satisfies the boundary and/or initial conditions and the other part solves the solution domain.

Let where is a Neural Network with at least one hidden layer, one input (), one output and all other trainable and non-trainable parameters (). For this type of machine learning problem, the cost function can be constructed in the following general form

|  |  |
| --- | --- |
|  | ***Eq. 1*** |

The last *Observed vs. Predicted Cost* componentis used when we have actual observations (typically noise) and in the inverse problem where in addition to the solution of the differential equation an attempt is made to fit the parameters simultaneously. In the forward problem sampling the solution domain is often sufficient.

***Example 1 – Classical Method***

Let us illustrate the classical method with an ordinary differential equation (*Eq. 2)*

|  |  |  |
| --- | --- | --- |
|  | | ***Eq. 2*** |
| *initial conditions* | , |  |

The solution of the second order differential equation in *Eq. 1* for the given parameter values and  is given in *Eq. 3*.

|  |  |
| --- | --- |
|  | ***Eq. 3*** |

The cost function is therefore

|  |
| --- |
|  |

***Example 2 – Transform Method***

As described by Lagaris2, one can carefully choose a solution form to satisfy the differential equation initial or boundary conditions; essentially allowing for an unconstrained optimization of the neural network.

This time let the solution of differential equation be approximated by where is a function of the time () input, and a neural network similar to the first example. More specifically, the psi () function can be written in the following manner

|  |  |
| --- | --- |
|  | ***Eq. 4*** |

The form of *Eq. 4*, is not unique but it satisfies both the initial conditions in *Eq. 2*.

At ,

|  |  |  |
| --- | --- | --- |
|  |  |  |

And

|  |  |
| --- | --- |
|  |  |

Accordingly, we can construct a simpler cost function this time since the initial conditions are already satisfied.

|  |  |
| --- | --- |
|  | ***Eq. 5*** |

***Example 3 – Case when initial and boundary conditions cannot be satisfied directly***

The same methods shown above may be extended for partial differential equations (PDE) with two or more dimensions. For example, given the following PDE in Eq. 6

|  |  |
| --- | --- |
|  | ***Eq. 6*** |

With initial and boundary conditions,

|  |  |
| --- | --- |
|  | ***Eq. 7*** |

This time will assume there are no real observation as included in the previous example.

For this specific problem, the author2 recommends a trial solution of the form below,

|  |  |
| --- | --- |
|  | ***Eq. 8*** |
|  |  |

Then we can verify that our initial and boundary conditions are satisfied

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

As shown, the last condition is not met unless the first and second derivative of the neural network with respect to its input evaluated at one are equal in value. The original author’s hypothesis is that the neural network can achieve this state. Nonetheless, this example illustrates the major shortcoming of this method as it is often difficult to produce a trial solution that satisfies all conditions.

In this case the simple cost function is written as follows,

|  |  |
| --- | --- |
|  | ***Eq. 9*** |
|  |  |

and the slightly modified version,

|  |  |
| --- | --- |
|  | ***Eq. 10*** |

Where lambda () is a small number which adds an additional cost to encourage the network to prioritize meeting the last mixed condition before minimizing the entire cost. *Eq.10* is introduced as a somewhat novel idea in this report and requires more testing to verify the results.

**The API**

*Section 1* in this report covered in detail reverse mode automatic differentiation which is perhaps the most under-rated machine learning algorithm at present. This library combined with Neural Networks provides a capable toolset able to solve differential equations among other problems. This section serves as a mini-tutorial in order to take advantage of the application programing interface (API) offered by TensorFlow and Keras.

The Functional API is one of three ways to construct a neural network architecture with Keras. *Table 1* provides a familiar example typical of common use.

**Table 1:** Keras Functional API example

|  |
| --- |
| # model parameters  batch\_size = 25  epochs     = 1000  # define model  inputs = keras.Input(shape=(2, ))  l1 = layers.Dense(10, activation="sigmoid")(inputs)  l2 = layers.Dense(10, activation="sigmoid")(l1)  l3 = layers.Dense(10, activation="sigmoid")(l2)  outputs = layers.Dense(1, activation="linear")(l3)  model = tf.keras.Model(inputs, outputs)  # print summary and compile  print(model.summary())  model.compile()  # create a dataset  xt\_ds = tf.data.Dataset.from\_tensor\_slices(xt\_train)  xt\_ds = xt\_ds.shuffle(len(xt\_train), seed=123)  xt\_ds = xt\_ds.batch(batch\_size)  xt\_ds = xt\_ds.cache()  # train model  model.fit(xt\_ds, epochs=epochs) |

Keras is a higher order library; therefore, the implementation details are hidden from the user. However, the most interesting part in Table 1 happens under the “fit()” method. In this method there are two loops where the outer one iterates over the epochs and the inner one iterates over the batches – this time over the dataset iterator which has already split and shuffled the data as specified by the user. The next step is to overwrite the training step inside this method to implement our own logic, i.e.: calculate gradients and update the architecture parameters. See *Table 2*.

**Table 2:** Extending the Keras model with custom training step

|  |
| --- |
| class ODENetwork(tf.keras.Model):      # define an optimization alforithm (gradient descent varient)      optimizer    = tf.keras.optimizers.Adam(learning\_rate=0.001)      # define a custom metric tracker      loss\_tracker = tf.keras.metrics.Mean(name="custom\_loss")      @tf.function      def loss\_f(self, x\_pred, dx\_dt, d2x\_dt2):          loss = tf.keras.backend.square(d2x\_dt2 + dx\_dt + x\_pred)          return loss      def train\_step(self, data):          # t is the nn-input (features)          # x is the observed value (labels)          x\_true, t = data          # record operations for automatic differentiation          # this captures second order derivatives          with tf.GradientTape(persistent=True) as tape\_ord\_2:                # explicitly watch non-trainable input              # trainable parameters are automatically watched              tape\_ord\_2.watch(t)                # record operations for automatic differentiation              # this captures first order derivatives              with tf.GradientTape(persistent=True) as tape\_ord\_1:                  tape\_ord\_1.watch(t)                  # apply forward step                  x\_pred = tf.cast(self(t, training=True), dtype=tf.double)                  # capture first and second derivative of the network                  # with respect to its input                  dx\_dt = tape\_ord\_1.gradient(x\_pred, t)                  d2x\_dt2 = tape\_ord\_2.gradient(dx\_dt, t)                  # calculate loss for each point in batch                  # returns loss for each point (important)                  loss\_f = tf.keras.backend.map\_fn(                      lambda x: self.loss\_u(x[0], x[1], x[2]),                      (x\_pred, dx\_dt, d2x\_dt2),                      dtype=tf.double)                    # average the loss column-wise;                  # hence, for 1-dim data does not do anything                  # gradients are averaged in the reduction api                  # this topic is not well documented in tf or keras                  loss\_f = tf.reduce\_mean(loss\_f, axis=-1)          # calculate gradients for all weights of the network and          # apply with the optimizer defined above          grads = tape\_ord\_1.gradient(loss\_f, self.trainable\_weights)          self.optimizer.apply\_gradients(zip(grads, self.trainable\_weights))            return {"custom\_loss": self.loss\_tracker.result()}      @property      def metrics(self):          return [self.loss\_tracker] |

All the magic is shown in Table 2. First, the Keras model is passed into the class argument, inheriting all the methods defined in that class object. Next and most importantly, the method called “train\_step()” is overridden in order to implement any custom logic. This method is called every batch step; therefore, it receives the batch data as an argument. It is very important to maintain the method name and the signature as specified above.

In the context of this report, the custom logic implements the following steps:

1. Record all gradients for the neural network with respect to its inputs
2. Calculate the loss for each point input in the batch data
3. Calculate gradients for all trainable parameters in the network with respect to the average loss in *Step 2*
4. And finally update the network parameters with stochastic gradient descent

**Table 3:** Usage of custom model

|  |
| --- |
| # typical model  model = tf.keras.Model(inputs, outputs)  # custom model  model = ODENetwork(inputs, outputs) |

Extending the typical usage of Tensorflow and Keras library is a must for any research effort. The template above is nicely packaged once again in the “fit()” method where we can still take advantage of the multi-worker training, passing existing or custom callback to track or troubleshoot the system, and all other features offered by the libraries. For completeness, Table 3 shows the usage of the Keras custom model. The main takeaway here is that a custom model allows one to create complex custom cost functions to solve non-standard problem such as differential equations.

**Discussion**

This report is intended to highlight the basic framework for solving differential equations, and therefore intentionally does not include statistics on the precision of the two methods covered in this work. This report starts with the description of perhaps the most important tool of this framework which is automatic differentiation. Although it is possible to analytically calculate the derivatives of a feed forward neural network, this method increases the search efficiency for a good architecture, and can also be applied to other problems which can be solved in a similar manner.

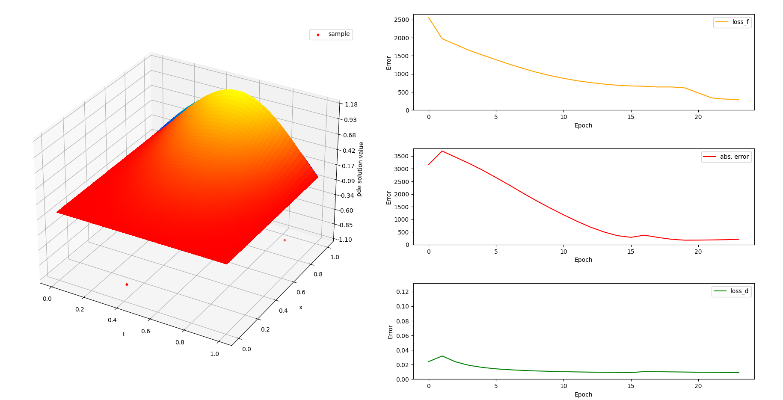
As stated in the methodology section, solving differential equations with a neural network can be thought of as a self-supervised method since the label zero arises from the mathematics of the problem (shifting the differential equations to one side sets these equations to zero). Then, as in many machine learning problems, the job of the cost function is to minimize the average sum – as close as possible to zero – and consequently solve the differential equations.

|  |  |
| --- | --- |
| **Figure 1:** ODE inverse problem training with the classic method and noisy data | **Figure 2**: ODE inverse problem training with the transform method and noisy data |

When comparing the two methods, the classical method is easier to apply and it is pretty reliable to obtain reliable results. In both of the differential equation examples treated in this report, the classical method obtains a smaller absolute error. Nevertheless, no matter how small the total error, it is almost impossible to fully satisfy the initial and boundary conditions. Alternatively, the transform method, when a trial solution can be formulated correctly, guarantees that the boundary conditions are satisfied at all times (*compare Fig 1 vs Fig 2, and Fig. 3 vs. Fig 5*). However, the problem with the transform method, at least from the experience gained from this work, is that it makes the stochastic descent more susceptible to getting stuck in local minima. In other words, the error manifold becomes more complicated.

|  |  |
| --- | --- |
| **Figure 3:** PDE training start with classical method | **Figure 4:** PDE training end with classical method |

The results for both methods can vary depending on the form of the equation. For example, the PDE considered in this report has a non-linear form (*see Eq. 6*) which made it very difficult to solve – particularly for the classical method. This is the reason why the transform method deserves much more attention that is has received in the literature. As shown in *Example 3,* even in the case where the analytical part of the trial solution does not fully satisfy all conditions, it is possible to constrain the neural network to overcome such difficulties. If there is a way to improve the training and overcome local minima, the transform method can become a reliable way to solve differential equations and other problems alike. At present additional research is necessary to improve the training and convergence of both these methods, and many other variations which are added in the literature every day.



***Figure 5:*** *PDE training with transform method*

**Conclusion**

In summary, this report highlighted the importance of automatic differentiating and the potential of this tool to solve many problems involving calculating gradients and derivatives. Two methods were introduced as a methodology to solve ordinary differential equations and partial differential equations while considering the forward problem and the highly sought-after inverse problem (fitting parameters). Both methods performed relatively well and were able to achieve respectable results given very little tunning of the architecture and hyperparameters. The transform method emerges as favorable because it offers to satisfy the boundary conditions, and the classic method proves to be very reliable to get a good basic solution. The neural network training was improved by introducing additional weights in the cost function to motivate the neural network to discriminate between the domain solution, boundary conditions, and point observation cost. Finally, this report presents one approach to apply both methods using the powerful API from TensorFlow and Keras. The full code for each of the four problems is included with the submission of this report.

**References**

1. *Reverse-mode automatic differentiation: a tutorial*

*[*[*https://rufflewind.com/2016-12-30/reverse-mode-automatic-differentiation*](https://rufflewind.com/2016-12-30/reverse-mode-automatic-differentiation)*]*

1. *Artificial Neural Networks for Solving Ordinary and Partial Differential Equations*

*[*[*https://arxiv.org/abs/physics/9705023*](https://arxiv.org/abs/physics/9705023)*]*