Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations

E4040.2020Fall.PINN.report
Gurpreet Singh Hora gh2546, Maria Katsidoniotaki mk4121
Columbia University

Abstract

With the advent of advanced computational resources and huge availability of the dataset, neural networks (NNs) have made significant progress in the field of cognitive sciences, image recognition, and genomics. Despite these great accomplishments, NNs still struggle to demonstrate their capabilities in the small data regime. To overcome the limitation, we introduced Physics-based neural networks (PINNs), the networks that are contrived for the complex physical and engineering systems, which can respect underlying laws of physics described by nonlinear partial differential equations (PDEs). Herein, we investigate PINNs that can be a potential surrogate models for approximating the PDEs. The versatility of the PINNs is based on the automatic differentiation technique, which allows us to incorporate the prior information i.e. the structure of the PDE in the network. This distinctive attribute of PINNs enabled it to efficiently approximate the classical problems representing nonlinear phenomena in the field of computational mechanics and computational stochastic dynamics using minimal training datasets. The main challenge was to build PINNs in TensorFlow 2 (originally written in TensorFlow 1) that corresponds to the uniqueness of the network architecture i.e. additional custom layers which represent the structure of the PDE. To resolve this, we took advantage of the GradientTape and tf.function utility which allows us to write and debug the code in the eager execution mode and execute as a TensorFlow graph. In this report, we provide two different approaches of code we built in TensorFlow 2 and demonstrate the success of the one code in achieving satisfactory results for Burger's and Schrodinger's equations.

1. Introduction

The Physics-informed neural networks (PINNs) presented herein are introduced in [1] where authors demonstrate the capability of NNs to address two main problems; to infer the solution of PDEs and to solve the inverse problem and system identification in continuous time and discrete time models. In the literature, we can find papers [1-5, 11, 12] that addresses the approximation of the PDE using NNs. The PINNs are crafted to learn the underlying laws of physics (initial and boundary conditions) by incorporating the structure of the PDEs into NNs using automatic differentiation. In that way the structure of the PDE acts as a prior information of the latent variables for the network and enables the network to learn the concepts with the modest/minimal training examples. As a result, the PDEs are efficiently approximated by the PINNs using minimal training datasets. This has significant computational contribution in complex physical and engineering problems such as fluids, quantum mechanics, reaction–diffusion systems, and the propagation of nonlinear shallow-water waves [2-5], as well as our own research in Computational Fluid Mechanics and Computational Stochastic Dynamics.

In the project, we are focusing on the first section of the original paper that is the data-driven solutions of the PDEs. The authors of the original paper implemented the PINNs using TensorFlow 1 [6] for several PDEs including Burger's and Schrödinger equations. They built the network layers in a naive way using for-loops as well as customized loss function accounting for different input datasets i.e. the initial and boundary conditions of the solution and the collocation points where we enforce the structure of the PDE. The structure of the PDE is acting as a regularizer in the network and due to this the authors did not employ any standard regularization technique. Our contribution in the project is to replicate the results of the original paper in Tensorflow 2 using flexible and robust using functional API formulation. In addition, we aim to take a step further by investigating the sensitivity of the model for different hyperparameters, i.e. activation functions, optimizers and learning rate.

The main challenge was to convert the network from the original naïve structure to an efficient automated API Neural Network which encompasses a stack of non-linear layers and automatic differentiation high-order derivative layers which arise from the PDEs and to perform end-to-end training. Another challenge was to build and compile the model with a customized loss function which computes the error between the network outputs and training data-sets of different size as mentioned above. Additionally, we also had to read the documentation of TensorFlow 1 to understand how tf.Session works because the original code was using it.

In the end we managed to have two different approaches to formulate the networks so we provide the pseudo-code of both. One of them was able to reproduce the same level of accuracy as the original PINNs and we present the PINNs results for two different numerical examples. Finally, due to the time limit we did not include additional examples from our research fields as stated in our report but this remains an open chapter for future work.

2. Summary of the Original Paper

2.1 Methodology of the Original Paper

The main goal of the paper [1] is that given the incomplete or partial measurements, how to infer the continuous spatio-temporal solution of the PDEs of the following form:

$$u_t + N[u] = 0,\ x \in \Omega, t \in [0, T], \quad (1)$$

where $u(t,x)$ is the latent variable of the PDE given as function of spatial and time variables, x and t respectively, $N[u]$ is a nonlinear differential operator, and Ω represents the domain and belongs to R^D space. Further, the left hand side of PDE can be expressed as the following function $f(t,x)$:

$$f := u_t + N[u], \quad (2)$$

The authors created two deep neural networks; the first network is a multilayer perceptron (MLP) network consisting of dense layers with *tanh* nonlinearity, to approximate the solution $u(t,x)$. The second network is a physics-informed network which is created by stacking non-linear dense layers and automatic differentiated partial differential equation layer PDE(λ) (Figure 1) to approximate the function $f(t,x)$. The key thing to notice here is that these two networks shared the same parameters and during the training we minimize the combined loss of $u(t,x)$ and $f(t,x)$. These two networks share the same parameters and trained by minimizing the cumulative mean square error;

$$MSE = MSE_u + MSE_f = \frac{1}{N_u}\sum_{i=1}^{N_u}\left|u(t_u^i, x_u^i) - u^i\right|^2 + \frac{1}{N_f}\sum_{i=1}^{N_f}\left|f(t_f^i, x_f^i)\right|^2 \quad (3)$$

where $\{t_u^i, x_u^i\}, i = 1, \ldots, N_u$ and $\{t_f^i, x_f^i\}, i = 1, \ldots, N_f$ are the initial & boundary points and collocation points respectively and u^i is the exact solution of the PDE (note that $f^i = 0$).

The authors built a relatively simple deep feed-forward neural networks architecture with hyperbolic tangent activation functions. They also used the custom loss function above because the number of initial/boundary points N_u and the collocation points N_f is not the same. Also, no additional regularization has been employed because the MSE_f works as a physics-informed regularizer according to author's empirical observations which leads the use of relatively small amount of training data N_u. In the previous papers [2-5] Neural Network was treated as a black-box but now the customized network approach of function and loss function has opened the so-called "black box" network. Lastly, they used the second-order L-BFGS optimization algorithm which belongs to the family of quasi-Newton methods to minimize the loss function and to train both networks in an end to end fashion.

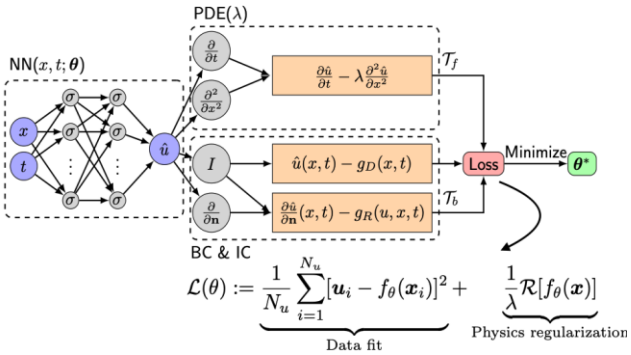


Figure 1 PINNs block-diagram from the original paper [14]

The block-diagram of the PINNs is provided in the Figure 1 above. The first block consists of the Neural Network $NN(t,x;\theta)$ which approximates latent variable $u(t,x)$. The second blocks $PDE(\lambda)$ consists of the derivative terms and along with $NN(t,x;\theta)$ it will approximate the PDE structure $f(t,x)$. Note that these two networks are sharing parameters.

2.2 Key Results of the Original Paper

Among all the numerical examples presented in the original paper, we worked the Burger's and Schrodinger equation and that is why we present them here.

2.2.1 Burger's equation

First, the authors illustrated how to employ PINNs for approximating the Burger's equation. Here, they demonstrated the ability of PINNs to capture the nonlinear higher order derivatives. Burger's equation along with Dirichlet boundary has the following formulation:

$$u_t + uu_x - \left(\frac{0.01}{\pi}\right)u_{xx} = 0, \quad (4)$$

where $x \in [-1, 1], t \in [0, 1]$,
$u(0,x) = -sin(\pi x)$
$u(t,-1) = u(t,1) = 0$

They built a 9-layer network with 20 neurons per hidden layer to approximate the latent variable solution as well as a custom layer of derivatives using automatic differentiation. In the training part, the authors used N_u=100 initial & boundary points and N_f=10,000 collocation points. The relative L_2-norm error of the predicted solution with the exact solution generated from Latin Hypercube Sampling strategy [3] is 6.7 10^{-4}. The results are demonstrated in Figure 2.

They also investigated the relative L_2-norm error for various combinations of initial and boundary points and

collocation points as well as for different numbers of layers and neurons to find the optimal number of training points, width and depth of the network. They confirmed that the accuracy increases as the number of data increases as well as the number of layers and neurons increases, according to Figures 3 and 4.

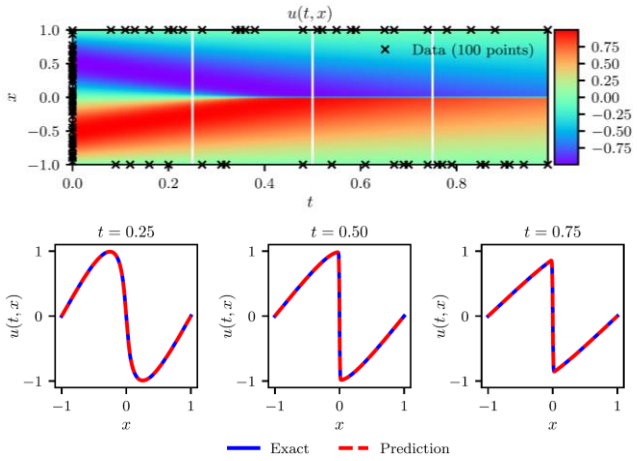


Figure 2 a) Predicted solution u(t, x) with 100 initial and boundary training data and 10,000 collocation points (from Latin Hypercube Sampling strategy). b) Comparison with the exact solution corresponding at the three time instances (white lines) depicted above. Relative L2-error = 6.7 ·10⁻⁴.

N_u \ N_f	2000	4000	6000	7000	8000	10000
20	2.9e−01	4.4e−01	8.9e−01	1.2e+00	9.9e−02	4.2e−02
40	6.5e−02	1.1e−02	5.0e−01	9.6e−03	4.6e−01	7.5e−02
60	3.6e−01	1.2e−02	1.7e−01	5.9e−03	1.9e−03	8.2e−03
80	5.5e−03	1.0e−03	3.2e−03	7.8e−03	4.9e−02	4.5e−03
100	6.6e−02	2.7e−01	7.2e−03	6.8e−04	2.2e−03	6.7e−04
200	1.5e−01	2.3e−03	8.2e−04	8.9e−04	6.1e−04	4.9e−04

Table 1 Relative L2-error for different number of initial and boundary training data N_u and collocation points N_f and fixed 9 layers with 20 neurons per hidden layer.

Layers \ Neurons	10	20	40
2	7.4e−02	5.3e−02	1.0e−01
4	3.0e−03	9.4e−04	6.4e−04
6	9.6e−03	1.3e−03	6.1e−04
8	2.5e−03	9.6e−04	5.6e−04

Table 2 Relative L2-error for different number of hidden layers and neurons per hidden layer and fixed number of training data Nu=100 and collocation points Nf=10,000.

2.2.2 Schrodinger equation

Next, the authors aimed to demonstrate the ability of PINNs to approximate complex-valued variables and to handle periodic boundary conditions using the Schrodinger equation. Schrodinger equation along with periodic boundary conditions is represented as:

$$ih_t + 0.5h_{xx} - |h|^2 h = 0, \quad (5)$$

where $x \in [-5, 5], t \in [0, \pi/2]$,
$h(0, x) = 2\,sech(x)$

$h(t, -5) = h(t, 5)$ *and* $h_x(t, -5) = h_x(t, 5)$

They built a 5-layer network with 100 neurons per hidden layer as well as a custom layer of derivatives using automatic differentiation to demonstrate the PDE network. In the training part, they acquired the high resolution dataset from the open-source package named Chebfun [4] with a spectral Fourier discretization with 256 modes and a fourth-order explicit Runge–Kutta temporal integrator with time-step $\Delta t = \pi/2 \cdot 10^{-6}$. The authors chose randomly $N_u = N_0 + N_b = 100$ initial & boundary points and $N_f = 20{,}000$ collocation points, all from the aforementioned high resolution dataset. The relative L_2-norm error of the approximated solution and the exact solution generated using Latin Hypercube Sampling strategy [3] is 1.97 10^{-3}. The results are demonstrated in Figure 5.

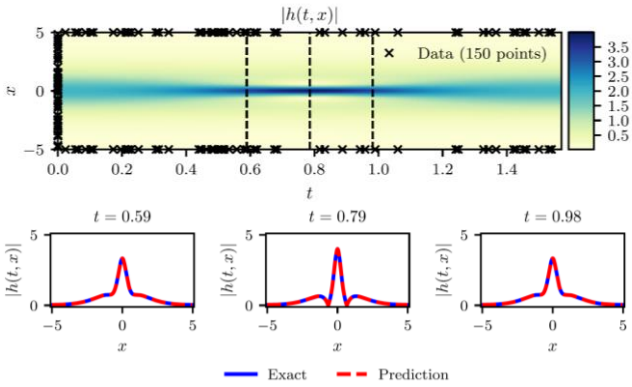


Figure 3 a) Predicted solution h(t, x) with 100 initial and boundary training data and 20,000 collocation points (from Latin Hypercube Sampling strategy). b) Comparison with the exact solution corresponding at the three time instances (white lines) depicted above. Relative L2-error = 1.97 10⁻³.

3. Methodology (of the Students' Project)

We will follow herein the same mathematical methodology of the original paper to solve PDEs of the form Eq. (1) and our main contribution of changes is in the coding part. Specifically, the original PINNs are modeled in TensorFlow 1 with customized layers as well as loss function. Even though it provides a good insight of the hidden part and avoids the "black box" solutions, we believe it lacks control and robustness. For instance, to change the activation function we need to change a big portion of the code compared to just changing an argument in the keras layers. Same thing is applicable for the Xavier initialization.

In the current project, we replicated the aforementioned results in TensorFlow 2 [10]. We leveraged Keras and TensorFlow and we chose Keras functional API to create PINNs because they are more flexible and can easily handle the multiple inputs or outputs. In addition to this,

we can add the custom layer of the derivatives in the network.

There are two main case studies presented in the original paper; (a) solution inference in continuous and in discrete time models, (b) parameter identification in continuous and in discrete time models. In our project, we reproduced the (a) solution inference in continuous time models and we worked on the two examples of Burgers and Schrodinger equation presented.

3.1. Objectives and Technical Challenges

In order to predict the latent variable $u(t,x)$ of the PDE in Eq. (1) which also respects the physical laws, we trained the network along with the structure of the PDE function $f(t,x)$ Eq. (2). Thus, we needed to account for high order derivatives of the network $u(t,x)$. In addition, the loss function in Eq. (3) accounted for different number of data; the initial and boundary points N_u and the collocation points N_f as it mentioned above.

We tried two different approaches; the first one was creating a Functional API network that takes the inputs x,t that represent the space and time domain respectively and produces two outputs, the solution $u(t,x)$ and the PDE $f(t,x)$. However, even if it was a clean and simple approach, we faced difficulties while compiling and fitting the model. Specifically, the network takes four inputs and predicts two outputs i.e. x and t from initial and boundary points N_u to calculate the latent variable $u(t,x)$ and x and t from collocation points N_f to calculate the PDE $f(t,x)$. However, the two outputs had different dimensions and therefore it was yielding dimensionality error to find the Mean Square Error while we were fitting the model, unless we have $N_u = N_f$. In addition, we were not able to include a customized loss function with this approach. Even if we did not manage to produce results, we will present the code in the next section.

In the second approach, we exploited the idea of Generative and Discriminative approach; we created a generative network that takes four inputs x,t that represent the space and time domain respectively for the two different number of datasets N_u and N_f and produces one output, the solution $u(t,x)$ and a discriminative network that predicts the PDE $f(t,x)$ based on the latent variable $u(t,x)$ only from the N_f collocation points. The main challenge we faced was to understand how tf.Session works. After studying the documentation of TensorFlow 1 and the codes of the original paper, we were successfully able to generate similar level of accuracy of the original paper.

3.2. Problem Formulation and Design Description

(a) First attempt

We created a Functional API network 'PINNs' that gets as two inputs $X = (x,t)$ that represent the space and time domain respectively and produces two outputs $Y = (u,f)$, where u the latent variable and f is the structure solution of the PDE and $f = u_t + uu_x - \lambda u_{xx}$. The block diagram is presented below.

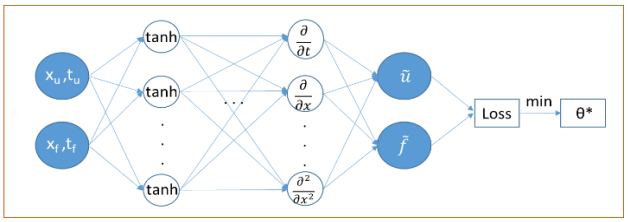


Figure 4 Block diagram for the PINNs network with four inputs and two outputs – Unfortunately, this does not work.

Here, we also present the pseudo-code of the network for the first example of Burgers equation; the network consists of 9 layers and 20 units for each hidden layer with hyperbolic tangent activation function:

```
def shared_model_func():
X = input(shape=[2,]) #space x and time t
xavier = Xavier Initialization
with tf.GradientTape() as g2:
   g2.watch(X)
   with tf.GradientTape() as g1:
      g1.watch(X)
      u = X
      for layer in range(8):
        u = Dense(20, 'tanh',xavier)(u)
      u = Dense(1, 'linear', xavier)(u)
    du = g1.gradient(u, X)
    u_x, u_t = du[:,0:1], du[:,1:2]
ddu = g2.gradient(du, X)
u_xx =ddu[:,0:1]
f=Lambda(u_t+u*u_x-lamda*u_xx)
model = Model(X, [u, f], name="PINNs")
return model
```

We were interested to calculate the solution $u(t_u,x_u)$ from the initial and boundary points N_u and to the PDE $f(t_f,x_f)$ from the collocation points N_f. Therefore, we make an additional function 'network' like that produces the outputs of our interest for the two different point sets. Based on that, we created an additional model 'PINNS_new' and this is what we aim to train:

```
def network(x_u, t_u, x_f, t_f):
model = shared_model_func()
X_u = tf.concat([x_u, t_u], axis=1)
[u_pred,f] = model(X_u)
X_f = tf.concat([x_f, t_f], axis=1)
[u,f_pred] = model(X_f)
return [u_pred, f_pred]
```

```
def create_network():
inps = [x_u, t_u, x_f, t_f]
outs = self.network(inps)
model=Model (inps, outs, name='PINNs_new')
return model
```

Lastly, we compile the model 'PINNS_new' and then we fit it to the training dataset to get the accuracy. The loss function between predicted data and training data is Mean Square Error. Then, we predicted the data from the testing dataset and compared the error.

```
model.compile(optimizer, loss = 'MSE')
inps = [x_u, t_u, x_f, t_f]
outs = [u, 0]
model.fit(inps, outs, epochs = eps)
u_pred, f_pred = model.predict(inps_test)
```

Unfortunately, the MSE between the network output $[u_{pred}, f_{pred}]$ and the data $[u, 0]$ is not feasible because the first cell of the output has dimension N_u and the second dimension N_f, unless $N_u = N_f$. On the other hand, we were getting syntax errors when we tried to use customized loss like the one below in the $model.fit$ that could not be overcome in the time frame of the project. Because of the simplicity of this first attempt, we present it here even if we could not get useful results.

```
def custom_loss(u_pred, u, f_pred):
loss = mean(square(u_pred- u),axis=-1) +
       mean(square(f_pred), axis=-1)
return loss
```

(b) Second attempt

We employed an MLP as a generator network which is approximating the latent variable $u(t,x)$ given the inputs x,t. The net_f (Figure 5) is used as a discriminator net which is guiding the net_u whether the approximated solution is correct or not. During the training phase, we calculate Mean Squared error loss i.e. MSE_u of $u(t,x)$ on initial and boundary conditions (red lines) and we calculate MSE_f of $f(t,x)$ on collocation points (green lines). The network presented with blue lines is a shared network between net_u and net_f.

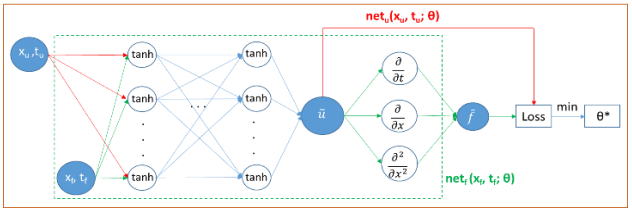


Figure 5 Block-diagram for the PINNs network we used to obtain the results.

Here, we created a network 'PINNS' that gets two inputs x,t and produces an output $Y = u$, where u the solution of the PDE. Again for the Burgers example, the 9-layer with 20 units network and hyperbolic tangent activation function is presented below:

```
def shared_model_func():
x = input(shape=[1]) #space x
t = input(shape=[1]) #time t
inp = concatenate([x,t])
xavier = Xavier Initialization
u = inp
for layer in range(8):
   u = Dense(20, 'tanh',xavier)(u)
u = Dense(1, 'linear', xavier)(u)
model = Model([x, t], [u], name="PINNs")
return model
```

The solution $u(t_u, x_u)$ for the initial and boundary points comes directly from the model. In order to define the PDE $f(t_f, x_f)$ as well as the customized loss that accounts for the MSE of both u and f;

```
def model_build_compile(x_f, t_f, x_u, t_u,
u, loss_function,optimizer):
with tf.GradientTape() as g2:
   u_pred = shared_model([x_u, t_u])
   loss_1 = loss_function(u, u_pred)
   with tf.GradientTape() as g1:
       g1.watch(x_f)
       g1.watch(t_f)
       u_f = shared_model([x_f, t_f])
       u_x = g1.gradient(u_f, x_f)
       u_t = g1.gradient(u_f, t_f)
    u_xx = g1.gradient(u_x, x_f)
    f_pred = u_t + u_f * u_x - lambda*u_xx
    loss_2 = loss_function(f_pred, 0)
    loss = loss_1 + loss_2
w = shared_model.trainable_weights
grads = g2.gradient(loss,w)
optimizer.apply_gradients(zip(grads, w))
return loss_1, loss_2, loss
```

where the loss function is the 'MSE' from the keras library. Therefore, the 'model_build_compile' is called at each step until the loss function is converged. This attempt gave us very satisfying results as indicated below.

We applied these codes for the two different examples presented in section 2 and the results as provided in Section 5.

4. Implementation

In the Section 4.1 we explained the Deep Learning architecture we are using to approximate PDEs. In the next Section 4.2 we are explaining the step-by-step implementation of the training and testing algorithm along with pseudo-code.

4.1. Deep Learning Network

We create a PINNs network (refer Figure 5) which consists of two sub-networks. First network is a multilayer perceptron network which has dense layers with tanh activation function. Second network is a custom network which consists of the derivative terms and created using automatic differentiation techniques. As shown in Figure 5 we are trying to minimize the combined loss: a.) loss on latent variable $u(t,x)$ given the initial and boundary conditions training data points b.) loss of the PDE $f(t,x)$ given the collocation training data set. We trained these two sub-networks in an end-to-end fashion.

To train the model, we used the double precision simulation data available on Github page [6]. Dataset was created by the authors of the original using the ChebFun package [8] and it is available on their GitHub page [6]. For the project, we used Burger's and Schrodinger equation simulation dataset. From the spatio-temporal data, we first created multiple arrays with the initial and boundary condition data and the collocation point data. Next, we randomly sampled the N_u points from the initial and boundary condition data to generate a training dataset for approximating latent variable $u(t,x)$ using net_u. Further, we selected N_f points for creating a collocation point training data set and to approximate PDE $f(t,x)$ using net_f. Lastly, we trained the PINNs using this entire training dataset. Note that in the original paper, the authors passed the entire dataset in an epoch instead of passing them in small batches. To keep uniformity, we adopted the same approach and passed the entire dataset. Next, we calculated the combined loss on variable $u(t,x)$ and PDE $f(t,x)$. Finally, we used GradientTape [10] and optimizer.apply_gradients [10] to minimize the loss function.

4.2. Software Design

Step-by-step process for training and testing the PINNs

Step 1: Load the data set. We implemented the *load_data()* function, which reads the data from mat prefix files and returns the spatial and temporal grid and the latent variable solution.
Step 2: Create a spatiotemporal grid using *mesh_grid()* function. Here, we pass the spatial and temporal discretization grid arrays as an argument to generate spatial and temporal coordinate arrays.
Step 3: Generate a test data array using *test_data()* functions. This function takes the spatial and temporal coordinate arrays and ground truth labels of latent variables $u(t,x)$ as an argument and returns the *X_test* and labels of the test data set.
Step 4: Generate a training data set using *training_data()*. This function takes spatialtemporal grid information, ground truth solution, lower and upper bound of the spatiotemporal domain and number of collocation points i.e. N_f. It will return the training labels of the latent variable $u(t,x)$ along with the initial and boundary condition dataset and input data to train net_f
Step 5: Next, we randomly select N_u points from the initial and boundary conditions arrays (mentioned above) to train net_u
Step 6: Then, we will use *shared_model_func()* to create a multilayer perceptron network. It will take list whose elements will be the number of units in the layer as an argument and returns a symbolic keras network.
Step 7: Create the optimizer from *custom_optimizer* function. It takes '*Adam/RMSprop/*' and its properties for instance, learning rate, decay step and so on.
Step 8: Create the Mean Squared Error loss function using *tf.keras.losses.MeanSquaredError()*
Step 9: We make a function *model_build_func()* for training the network. It takes input training data, training labels, shared model, loss function, optimizer. Inside this function we are calculating losses and updating the weights of the network.
Step 10: We repeat Step 9 until the loss curve becomes flat.
Step 11: We used the trained model for predicting variable $u(t,x)$ on the test data set using *predict()* function

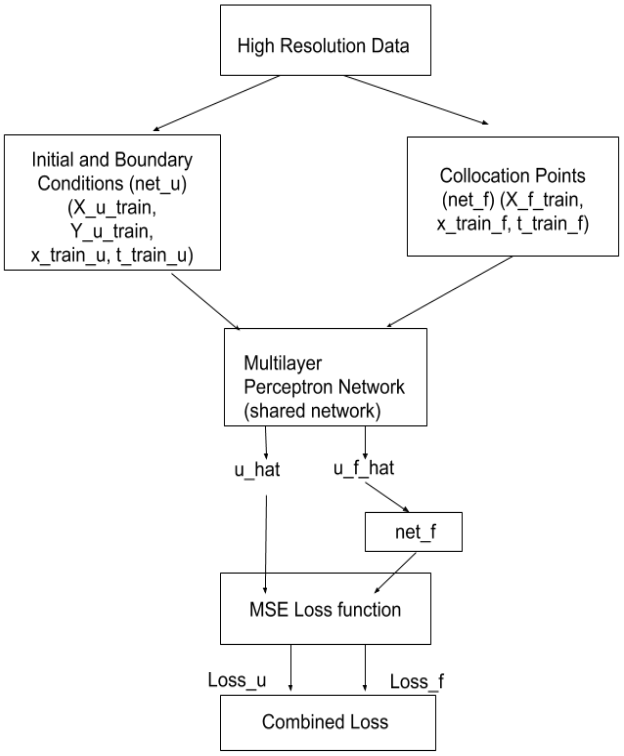


Figure 6 Flow chart representing the training phase of the PINNs

Pseudo code for each section of the implementation

```
## Require the following ##
Dataset; # High Resolution Dataset
N_u; # Number of initial and boundary
conditions training points
layers, units #Number of layers and units in
each layer
N_f; # Number of collocation points
lr, ds, dr; # learning rate, decay steps,
decay rate
num_epoch; # Number of epochs

## Extract the spatial grid (x), temporal
information (t), and spatio-temporal
solution (u) ##

## Spatial Temporal Grid ##
X, T = meshgrid(x, t);
# Domain Bounds
lb, ub = [X.min, T.min], [X.max, T.max];

## Prepare Training Data ##
# inputs to train net_u
inp_x_u_train = vstack([X_IC, X_BC]);
inp_t_u_train = vstack([T_IC, T_BC]);
y_u_train = vstack([u_IC, u_BC]);

## Generate a latin hypercube design (lhs)##
# inputs to train the net_f
X_f_train = lb + (ub - lb) * lhs(2, N_f);
inp_x_f_train = X_f_train[:,0:1];
inp_t_f_train = X_f_train[:,1:2];

## Prepare Test Data ##
X_test = hstack([X.flatten(), T.flatten()])
y_test = u;

# Optimizer
optimizer =choose_optimizer(lr, ds, dr);
# Loss function
MSE = MeanSquaredError();
# MLP Network
shared_model = shared_model_func(layers);

# Build and compile:
def build_compile():
with GradientTape() as tape_out:
   u_pred = shared_model([inp_x_u_train,
            inp_t_u_tran]);
   u_f = shared_model([inp_x_f_train,
       inp_t_f_train]);
   # Now adding custom layer #
   with GradientTape as deriv_tape:
   #keep them inside the loop because we
   want them to be recorded on tape#
      u_x = deriv_tape(u_f, x_f);
      u_t = deriv_tape(u_f, t_f);
   u_xx = deriv_tape(u_x, x_f);
   u_tt = deriv_tape(u_t, t_f);
   f_pred = c1 * u_t + c2 * u_xx + c3 *
            u_tt + c4 * u * u_x;
   MES_u = MSE(u_pred, y_u_train);
   MSE_f = MSE(f_pred, 0);
   custom_loss = MES_u + MSE_f;

## To minimize the custom loss and updates
the weights ##

variables = shared_model.trainable_weights
grads = tape_out.gradient(custom_loss,
variables);

return costum_loss

#Training
for epoch in range(num_epochs):
   loss.append(build_compile);

# Test Phase
x_test = X_test[:,0:1];
t_test = X_test[:,1:2];
u_hat = shared_model.predict([x_test,
      t_test]);
```

All codes can be found in *utils* folder https://github.com/orgs/ecbme4040/teams/pinn

5. Results

5.1. Project Results

5.1.1 Burger's equation

a) Hyperparameters Tuning

A suite of numerical experiments is carried out on the aforementioned architecture where net_f represents the structure of the Burger equation and corresponding training dataset, and test dataset. We programmatically vary the optimizer algorithm, learning rate aka step size, depth of the network, and the number of units in each layer. For this analysis, we designed a random grid by considering 3 optimizer algorithms i.e. Adam, RMSprop, and SGD, 4 randomly generated learning rates in the range (0.001, 0.1), 4 networks with different depths i.e. 4, 6, 8, 10 and 3 networks with different number of units i.e. 10, 20, 30.

To perform the analysis, we explored all the possible combinations of these 4 variables and carried out 144 numerical experiments. We have presented the best 5 results in Table 3. From the results, it can be seen that Adam has outperformed other optimization algorithms. In addition to this, we also noticed that RMSprop gives reasonable accuracy and SGD performed very poorly. We obtained 0.0098 as the best learning rate. Networks of size (depth, width) = (8, 10) and (6,20) give acceptable results. On the basis of this analysis, we chose Adam algorithm as an optimizer with learning rate of 0.01 and 0.95 decay rate in order to get close to 0.0098. The selected network architecture was 9 layers and 20 units in hidden layer.

Once we finalize our optimizer, learning rate and the architecture of the network, we varied activation functions as we suggested in the proposal. Results for various

activation functions are reported in Table 4. Our investigation suggests that our neural network makes good predictions if we employ sigmoid and tanh activation functions and fail to perform with relu or leaky relu functions. Also, this resembles with the ideal trend i.e. Relu and its variation may not perform for the dataset containing negative values. Here, our spatial grid is centered around origin and encompasses negative coordinate values. This may be a potential reason why Relu does not work in our case. At the end, we trained our Burger's PINNs with Adam optimizer, learning rate of 0.01 with a decay rate of 0.955, tanh activation function.

Label	Optimizer	Step size ϵ	Layers	Neurons	Error
1	Adam	0.0098	8	10	0.10
2	Adam	0.0098	6	20	0.17
3	Adam	0.0098	6	10	0.17
4	Adam	0.0397	4	10	0.22
5	Adam	0.0098	10	10	0.29

Table 3 The 5 best results obtained by the hyperparameter analysis of the Burger's equation for a fixed 9-layers and 20 units in each hidden layer architecture.

Activation Function	Error
Sigmoid	$3.27 \cdot 10^{-2}$
Tanh	$1.42 \cdot 10^{-2}$
Relu	N/A

Table 4 Investigation of various activation functions for Burger's PINNs for a fixed 9-layers and 20 units in each hidden layer architecture with other hyperparameters kept fixed..

b) Results

In Table 5, we presented the relative L_2-error for the 12 different neural network architectures with the abovementioned optimizer, learning rate, activation function. In this analysis, we vary the width and depth of the network. We found that the relative L_2 norm gets smaller as we increase the number of layers and number of units. To avoid overfitting/underfitting and at the same time to keep our model as simple as possible, we selected the network with 9 layers and 20 units in each hidden layer. This also resembles the architecture of the original paper.

In Figure 9, we have plotted the loss curves for the latent variable $u(t,x)$, PDE $f(t,x)$, and the combined loss of the model as a function of number of epochs. From the loss history, we can observe that our loss has converged and curves become flat. This means that our network is trained and learnt the nonlinear behaviour of Burger's PDE. In Figure 7, we plotted the prediction made by our trained network. On the top, we plotted the spatial-temporal approximated solution and on the bottom, we compared the predicted and ground truth solution at three different instantaneous snapshots. We can see that our predicted solutions are in a very good agreement with the ground truth data. To get more insights of the difference between predicted and ground truth solution, we plotted the spatio-temporal error in Figure 8. In the error heat map, we can infer that our network is able to predict latent variables in the entire domain except around origin. This may be attributed to the sharp discontinuity in the latent variable as we move along the time axis.

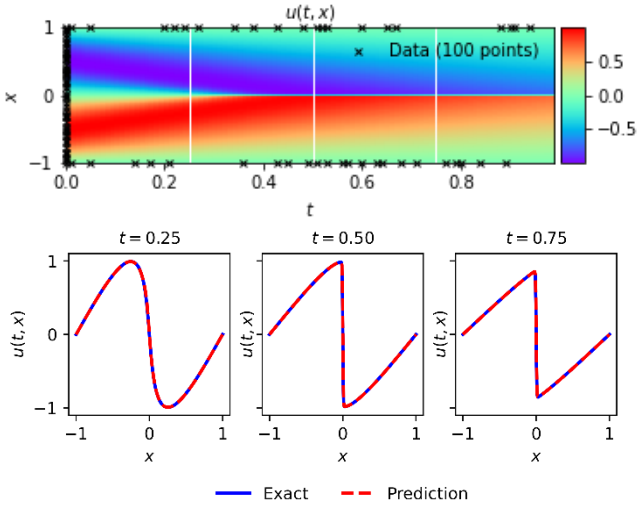


Figure 7 a) Predicted solution u(t, x) with 100 initial and boundary training data and 10,000 collocation points (from Latin Hypercube Sampling strategy). b) Comparison with the exact solution corresponding at the three time instances (white lines) depicted above. Relative L_2-error = 1.42 10-2.

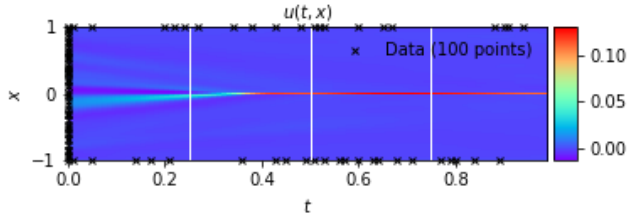


Figure 8 The error heat map of the solution u(t, x) with 100 initial and boundary training data and 10,000 collocation points.

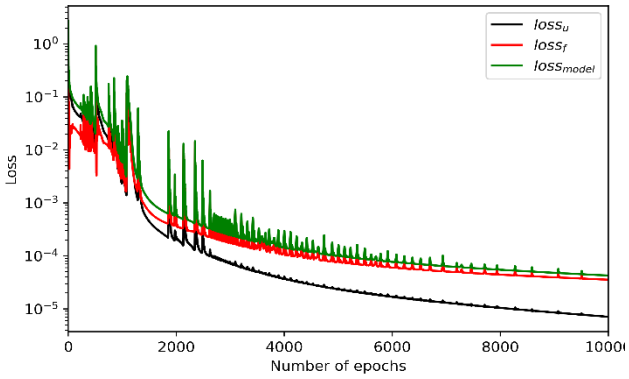


Figure 9 The loss curve of the network for 10,000 epochs.

Nu / Nf	2000	6000	10,000
20	$4.05\ 10^{-1}$	$2.31\ 10^{-1}$	$9.29\ 10^{-1}$
40	$7.20\ 10^{-1}$	$4.45\ 10^{-2}$	$2.92\ 10^{-2}$
60	$2.68\ 10^{-1}$	$6.33\ 10^{-2}$	$2.60\ 10^{-2}$
80	$5.85\ 10^{-1}$	$2.77\ 10^{-2}$	$1.43\ 10^{-2}$
100	$2.13\ 10^{-1}$	$1.07\ 10^{-2}$	$1.42\ 10^{-2}$
200	$3.19\ 10^{-2}$	$2.84\ 10^{-2}$	$1.34\ 10^{-2}$

Table 5 Relative L_2-error for different number of initial and boundary training data N_u and collocation points N_f and fixed 9 layers with 20 neurons per hidden layer.

Neurons / Layers	10	20	40
2	$4.31\ 10^{-1}$	$2.49\ 10^{-1}$	$4.22\ 10^{-1}$
4	$8.60\ 10^{-2}$	$9.54\ 10^{-2}$	$3.58\ 10^{-2}$
6	$4.97\ 10^{-2}$	$2.99\ 10^{-2}$	$6.29\ 10^{-3}$
8	$6.22\ 10^{-2}$	$9.85\ 10^{-3}$	$6.78\ 10^{-3}$

Table 6 Relative L_2-error for different number of hidden layers and neurons per hidden layer and fixed number of training data N_u =100 and collocation points N_f =10,000.

5.1.2 Schrodinger equation

a) Hyperparameters Tuning

As described above, for the tuning of the hyperparameters, we did an exhaustive random search on 3 pairs of optimizers, 3 pairs of learning rate, 3 pairs of depth of the network, and 3 pairs of width of the network. In total we carried out 81 experiments by making all possible combinations of hyperparameters and results for 5 best experiments are reported in Table7.

From the analysis, we observed that Adam outperforms RMSprop and SGD optimizer algorithm. The best suited learning rate we obtained is 0.0004 and the best architectures we got are all possible combinations of depth values 4, 6, 8 and width 10. On the basis of this analysis, we finalized Adam optimizer algorithm to train our model. To keep our network architecture the same as the architecture defined in the original paper, we kept the number of layers as 5 and number of units as 100. For training the model, we first adopted the learning rate as 0.0004 and we observed lots of fluctuations in the loss value. We suspected that due to these fluctuations we may not be able to find the global minima of the function. Next, we trained our model again with a small learning rate 0.0003 and this time we also turned on the decaying of the learning rate and we managed to reduce the fluctuations in the loss curve.

Label	Optimizer	Step size ϵ	Layers	Neurons	Error
1	Adam	0.0004	6	100	0.43
2	Adam	0.0004	6	200	0.44
3	Adam	0.0004	4	100	0.45
4	Adam	0.0038	6	100	0.45
5	Adam	0.0038	4	100	0.49

Table 7 The 5 best results obtained by the hyperparameter analysis of the Schrodinger equation for a fixed 5-layers and 100 units in each hidden layer architecture.

b) Results

In Figure 12, we have plotted the loss curve for the real part (u) and imaginary part (v) of the latent variables. The subscripts IC, BC, and x represent the initial condition, boundary conditions, and derivative of the variable with respect to x. On the top of the figure, we plotted the loss in each individual term and due to multiple loss curves we are not able to draw interpretation from it. On the bottom, we plotted the combined loss curve i.e. the loss history of the model. The convergence of loss history suggests that our model has learnt the nonlinear behavior of Schrodinger's equation. In the curve, we can see fluctuations in the loss curve. It may be due to the fact that we plotted all these results in the log scale and it amplifies small values. In Figure 10, we plotted the magnitude of the latent variables

varying with time and space (on the top) and comparison of predicted solution with the ground truth solution (on the bottom). The results seem in a reasonable agreement with the ground truth results. Here, again we can see some discrepancies in the results around the origin and our network is not able to predict the sharp rise in the solution around the origin. We also plotted spatial-temporal error in Figure 11. We can see that our error is high around origin and at time 0.8 sec. This is the region where ground truth has sharp rises. Due to time limits and computational resources restrictions, we did not try different network architectures or activation functions for this example.

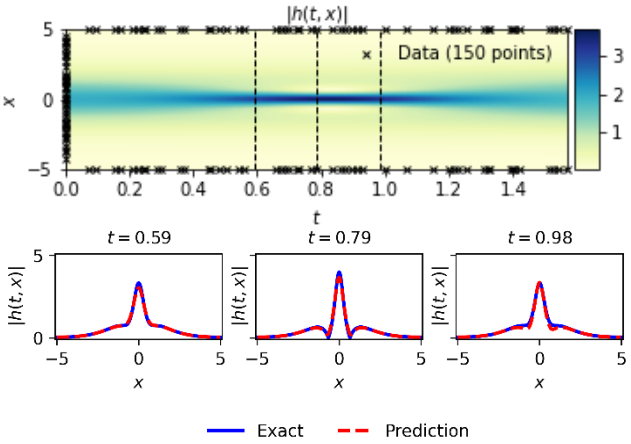


Figure 10 a) Predicted solution u(t, x) with 100 initial and boundary training data and 20,000 collocation points (from Latin Hypercube Sampling strategy). b) Comparison with the exact solution corresponding at the three time instances (white lines) depicted above. Relative L_2-error = 6.68 10-2.

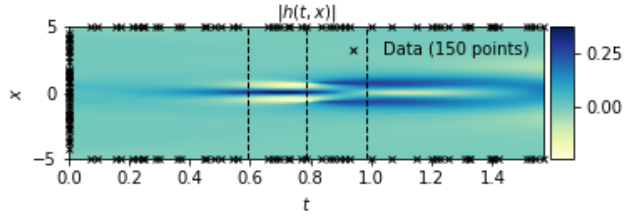


Figure 11 The error heat map of the solution u(t, x) with 100 initial and boundary training data and 20,000 collocation points.

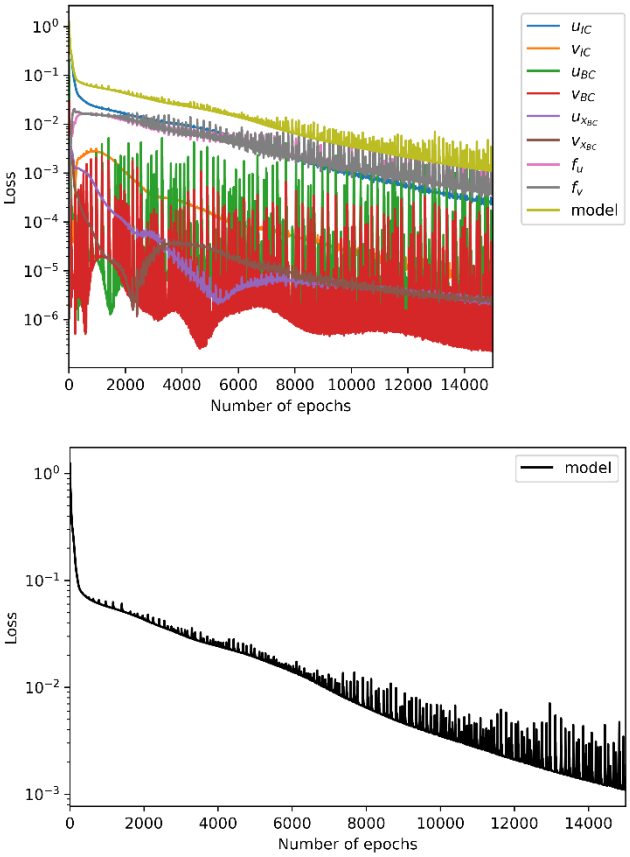


Figure 12 The loss curve of the network for 15,000 epochs.

5.2. Comparison of the Results Between the Original Paper and Students' Project

In Section 2, we presented the results of the original paper and in Section 5, we presented the results of our own PINNs. For the Burger's equation, if we perform the visual inspection of Figure 2 and Figure 7, our results are almost identical to the original paper. We have some very small discrepancies in the relative L_2 norm of the solutions presented in Table 1, 2, 5, and 6.

In the same way, for the Schrodinger equations, if we perform the visual inspection of Figure 5 and Figure 10, we can see our results are almost identical to the original results. Despite this, we have some discrepancies in the relative L_2 norm error of the solution.

In the original paper, authors did not provide any information of the loss curves so we are not in the position to compare our loss curves with them. But, to make sure our loss functions are good, we compared them with other network architectures named "Conserved physics-informed neural networks (cPINNs) and Extended physics-informed neural networks in (xPINNs)" presented in [11, 12]. We found that they have more oscillations in the loss history as compared to ours. Our best educated guess is that the authors did use the large learning rate.

5.3. Discussion of Insights Gained

As it is mentioned above, our results are in a very good agreement with the results presented in the original paper. There are some discrepancies in the results. These may be due to the fact that the original paper has used the 'L-BFGS' optimization algorithm while we used the Adam optimization algorithm. We chose Adam, RMSprop, and SGD (last two are used for hyperparameters tuning) over the second order 'L-BFGS' because we want to replicate the results with first order algorithms. This investigation also helps us in getting to know what other options of optimization algorithms are available for the Computational Machine Learning community to train there network. Another possible reason for this discrepancy is the different learning rate. In the original paper, they did not explicitly mention the learning rate. For our PINNs, we selected the learning rate based on our exhaustive hyperparameter search numerical experiment. Last but not least reason can be the datatype. For our PINNs we kept the data type as float 64 and in the original PINNs the authors kept it as float 32. We chose float 64 in order to keep the same data type of network as well as the training and test data set.

6. Conclusion

We have investigated physics-informed neural networks (PINNs), a potential surrogate model for solving the PDEs. These deliberately engineered neural networks are capable of capturing and learning the physical laws hidden in the dataset. We have explained how the structure of the PDE can be incorporated into the neural network using the automatic differentiation technique. We illustrated the application of PINNs on two PDEs equations named Burger's and Schrodinger equations. We found that our networks are able to learn the concept with the modest number of training data points. Using this, we demonstrated that PDE structure is acting as a prior information and a regularizing agent in the neural network.

In addition to this, we explored different optimization algorithms, activation functions, and different network architectures to develop a small database of potential hyperparameters for the Computational Machine Learning community. For the 1-D PDEs with real and complex valued latent variables, we are able to capture the nonlinear behavior using a relatively small neural network. Due to small network architecture, we did not employ any lasso, ridge or dropout regularization techniques. In the future, we are planning to extend this framework to solve 2-D and 3-D PDEs where our model architecture will be very large and during training of these huge architectures we may need more traditional regularization techniques.

6. Acknowledgement

We would like to thank Professor Zoran Kostic for teaching the Neural Networks class this semester and giving us the basic concept and theory behind it. We would also like to thank the Teaching Assistants, especially Desmond, for their helpful recitations and Office hours as well as their help whenever we needed it through additional meetings and emails.

7. References

[1] M. Raissi, P. Perdikaris, G. E. Karniadakis, "Physics - informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations", *Journal of Computational Physics 378* (2019) pp 686–707.

[2] J.X. Wang, J. Wu, J. Ling, G. Iaccarino, H. Xiao, "A comprehensive physics-informed machine learning framework for predictive turbulence modeling" (2017) *arXiv*:1701.07102.

[3] Y. Zhu, N. Zabaras, "Bayesian deep convolutional encoder-decoder networks for surrogate modeling and uncertainty quantification" (2018) *arXiv*:1801.06879.

[4] T. Hagge, P. Stinis, E. Yeung, A.M. Tartakovsky, "Solving differential equations with unknown constitutive relations as recurrent neural networks" (2017) *arXiv*:1710 .02242.

[5] R. Tripathy, I. Bilionis, "Deep UQ: learning deep neural network surrogate models for high dimensional uncertainty quantification" (2018) *arXiv*:1802 .00850.

[6] M. Raissi, PINNs (2017) GitHub repository https://github.com/maziarraissi/PINNs

[7] M. Stein, "Large sample properties of simulations using Latin hypercube sampling", *Technometrics 29* (1987) pp 143–151.

[8] T.A. Driscoll, N. Hale, L.N. Trefethen, "Chebfun Guide" (2014)

[9] "PINNs for 1D Burgers Equation(TF2.0).ipynb" Pierre Jacquier,https://colab.research.google.com/drive/1lo7Kf8 zTb-DF_MjkO8Y07sYELnX3BNUR

[10] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ... & Kudlur, M. (2016). Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)* (pp. 265-283).

[11] Jagtap, A. D., Kharazmi, E., & Karniadakis, G. E. (2020). "Conservative physics-informed neural networks on discrete domains for conservation laws: Applications to forward and inverse problems". *Computer Methods in Applied Mechanics and Engineering*, *365*, 113028.

[12] Jagtap, A. D., & Karniadakis, G. E. "Extended Physics-Informed Neural Networks (XPINNs): A Generalized Space-Time Domain Decomposition Based Deep Learning Framework for Nonlinear Partial Differential Equations" (2020) *Communications in Computational Physics*, 28(5), 2002-2041

[13] Helon Ayala, "Deep learning of vortex-induced vibrations", PINN, (2020), GitHub repository, https://github.com/helonayala/pinn

[14] CIS522: Deep learning

https://www.seas.upenn.edu/~cis522/slides/CIS522_Lecture11T.pdf

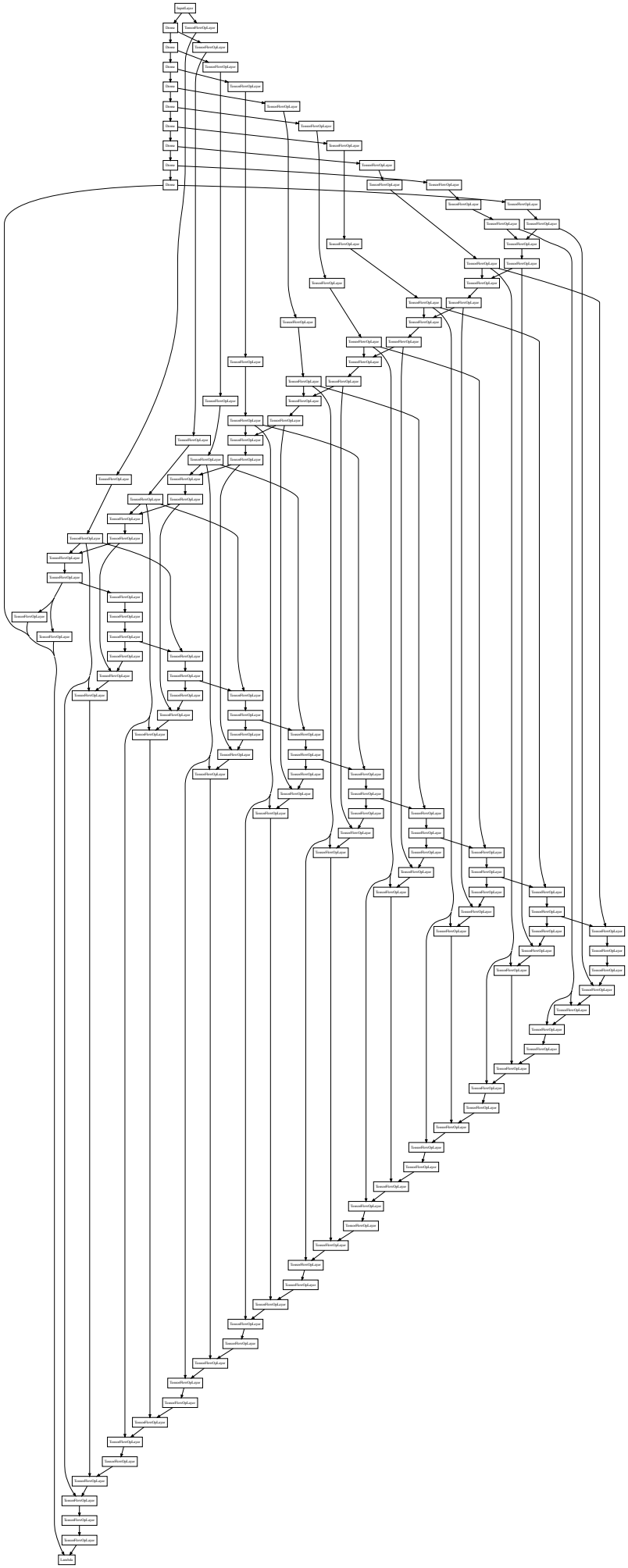
8. Appendix

8.1 Individual Student Contributions in Fractions

	UNI1 gh2546	UNI2 mk4121
Last Name	Hora	Katsidoniotaki
Fraction of (useful) total contribution	1/2	1/2
What I did 1	I found the orginal paper to work on. I implemented the first ideas of the coding part. We worked together with Maria to implement the two approaches above. When the first attempt failed, I came up with the second attempt idea. Once we decided the final algorithm, I plotted all the graphs and figures.	
What I did 2	I worked with Gurpreet in coding part to implement the network and try to solve the syntax errors especially in gradients. Once we decided the final algorithm, I organized the report and we updated together for its final formulation.	

8.1 Additional material

The flow chart of the first approach presented in section 3.2 is presented in the next page.


InputLayer
Dense
TensorFlowOpLayer
Concat