MIT Supervisor Neural Networks Direction Project Summary Report

Junxiao Zhao

New York University

26/7/2021

Catalogue

1. Project Background

1.1 What are Artificial Neural Networks?

Artificial Neural Networks (ANNs) are algorithmic mathematical models that mimic the behavior characteristics of animal neural networks and processes distributed and parallel information. Depending on the complexity of the system, this kind of networks can process information by adjusting the interconnecting relationship between a large number of internal nodes.

1.2 Development and Applications of Artificial Neural Networks

In 1943, Warren McCulloch, a neurophysiologist and cybernetician, and Walter Pitts, a logician who worked in the field of computational neuroscience, developed the first artificial neural network to mimic the operations of neurons in animal brains.

In 1958, Psychologist Frank Rosenblatt created the "Perceptron" model, which was a multi-layer neural network. This work was the first to put the study of artificial neural networks from theoretical discussion into engineering practice.

In 1975, social scientist and machine learning pioneer Paul Werbos developed the Back Propagation to train the Artificial Neutral Networks (Joshi 4)[1].

And now, with the continuous improvement of the algorithm and the increase of computing power and data, Neural Networks has been used in pattern recognition, automatic control, signal processing, decision aid, artificial intelligence, automatic driving, and so on.

[1] Joshi, Naveen. "The evolution of neural networks." *Allerin*. Sunday 16, Dec 2018. https://www.allerin.com/blog/the-evolution-of-neural-networks

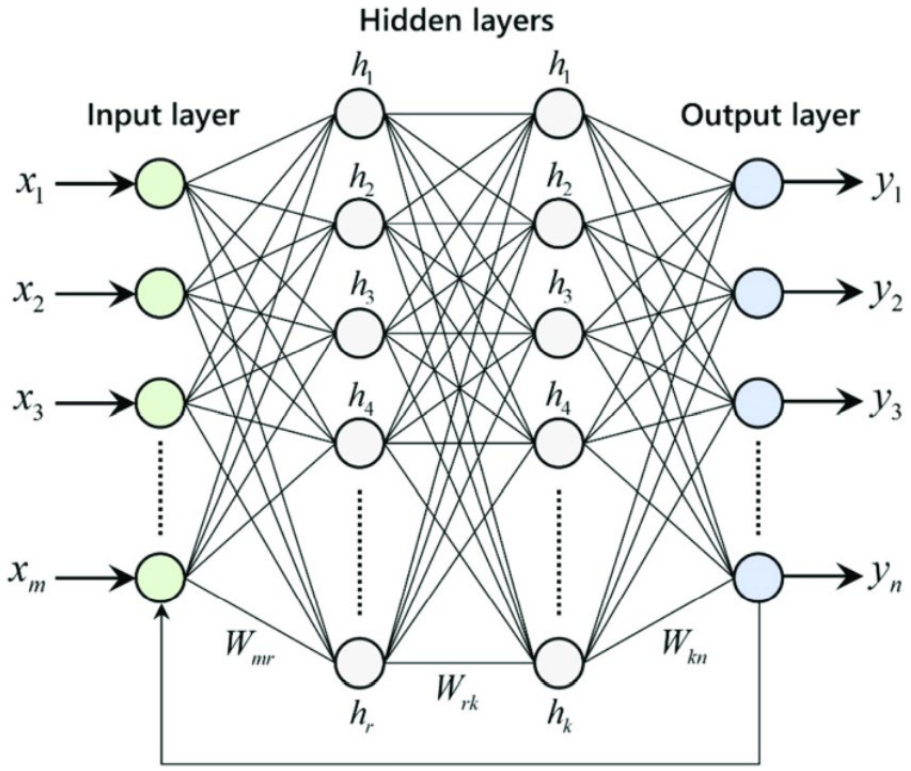
2. Project Process

2.1 Artificial Neural Networks

2.1.1 The constitutions and operation principles of Neural Networks

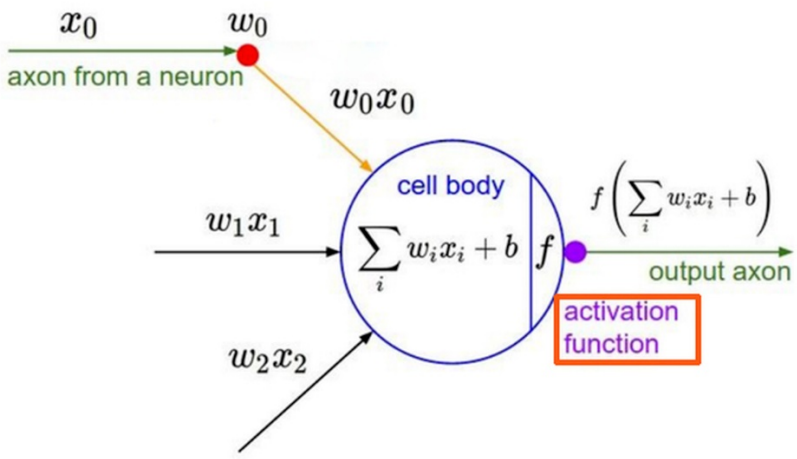
Structure

A neutral network contains three parts: an input layer, n hidden layers and an output layer, and each layer is constituted of connected units or nodes called artificial neurons.



Artificial Neurons and Weights

Basically, an artificial neuron receives a signal, computes it, and passes it to the connected neurons in the next layer. The connections between neurons are called “edges,” and each edge contains different weights that adjusts as learning proceeds, which increase or decrease the strength of the signals at connections.

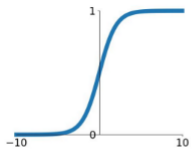


Activation Functions

After summing the weights times the signals from the previous layer and before passing it to the next layer, the output is computed by some non-linear functions called activation functions. Usually, we use tanh(x), sigmoid(x), relu(x), etc. as activation functions. The reason why we use the non-linear activation functions is that they are used to add nonlinear factors to improve the expression ability of the neural networks to solve the problems that the linear model cannot solve. Also, if the activation functions are linear, n hidden layers could be replaced by only one layer.

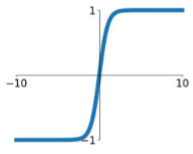
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



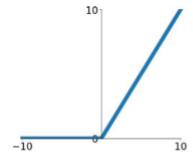
tanh

$$\tanh(x)$$



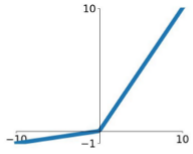
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

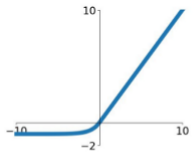


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

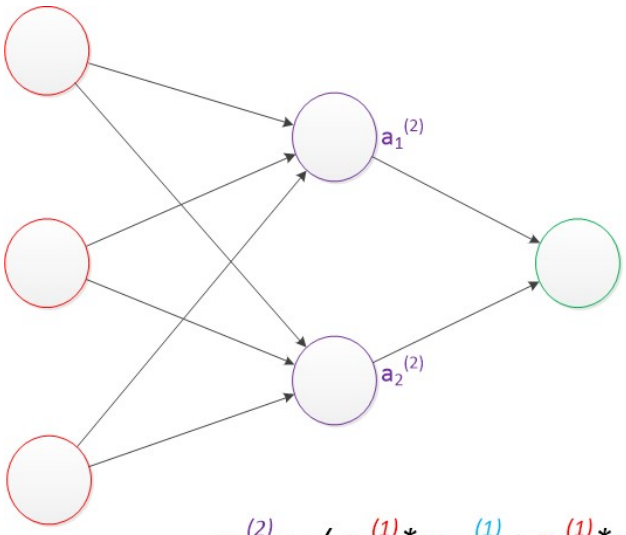
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Calculation

By adding and multiplying layer by layer, we end up with an output, and this process can be abbreviated as matrix multiplications.



$$a_1^{(2)} = g(a_1^{(1)} * w_{1,1}^{(1)} + a_2^{(1)} * w_{1,2}^{(1)} + a_3^{(1)} * w_{1,3}^{(1)})$$

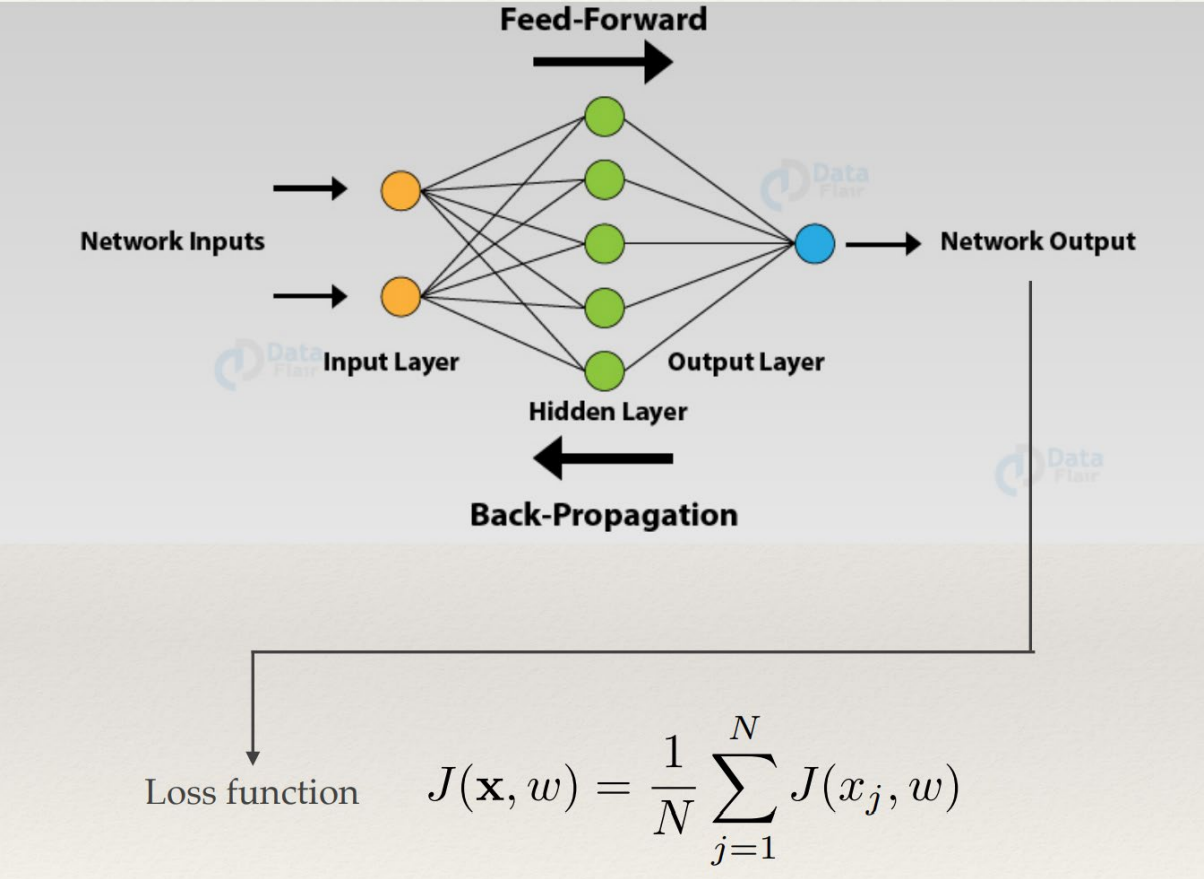
$$a_2^{(2)} = g(a_1^{(1)} * w_{2,1}^{(1)} + a_2^{(1)} * w_{2,2}^{(1)} + a_3^{(1)} * w_{2,3}^{(1)})$$

$$\begin{pmatrix} a_1^{(2)} \\ a_2^{(2)} \end{pmatrix} = g\left(\begin{pmatrix} w_{1,1}^{(1)} & w_{1,2}^{(1)} & w_{1,3}^{(1)} \\ w_{2,1}^{(1)} & w_{2.2}^{(1)} & w_{2,3}^{(1)} \end{pmatrix} * \begin{pmatrix} a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \end{pmatrix} \right)$$

But because initially the weights and bias are generated randomly, there is still a gap between the output and the true value. In order to use the model to predict the true value, the weights and bias on edges need to be modified.

Loss Functions

In order to calculate the accuracy of the model prediction, we introduce the loss function.



While performing the Linear regression (curve fitting), we usually choose Mean Square Error as the loss function, which calculates the square of the distance between the predicted value and the true value. The smaller the loss, the better the prediction.

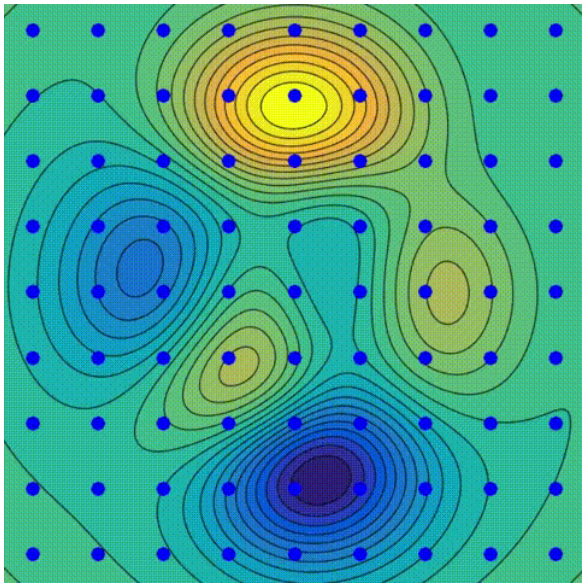
Mean squared error

$$loss_d = [u(0) - u_0]^2$$

Optimization and Gradient descent

After calculating the loss, we need to modify the weights and bias to optimize the prediction. In order to minimize the loss, we use gradient descendent, a first-order iterative optimization algorithm for finding a local minimum of a differentiable function. The direction of gradient increase is obtained by calculating the derivative of the loss function and each weight and bias. So

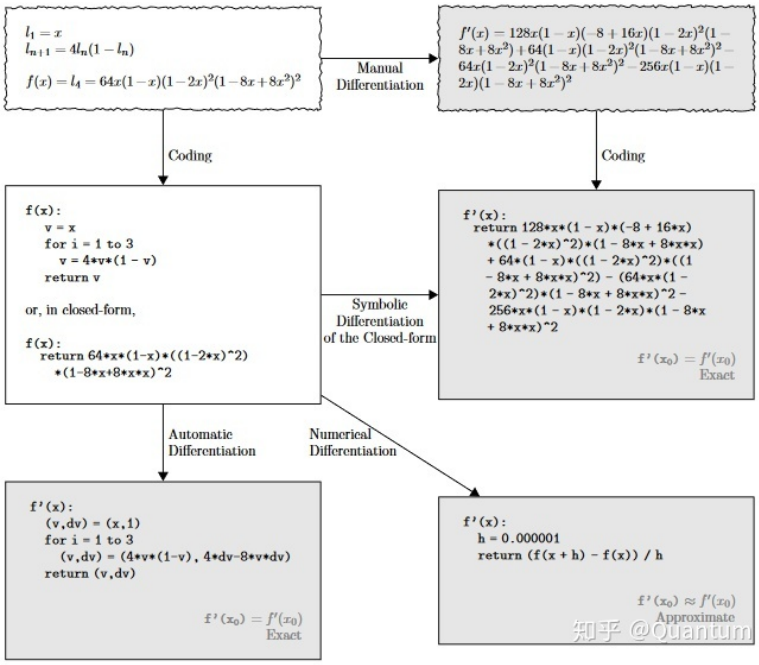
by subtracting the derivative times the learning rate, each weight and bias could adjust its value to help the loss to reach the local minimum.



Optimization $$w^{(i+1)} = w^{(i)} - \eta^{(i)} \nabla_w J(\mathbf{x}, w^{(i)})$$

2.1.2 Differentiations

There are mainly 3 methods of differentiation, which are Manual or Symbolic, Numerical and Automatic Differentiations.

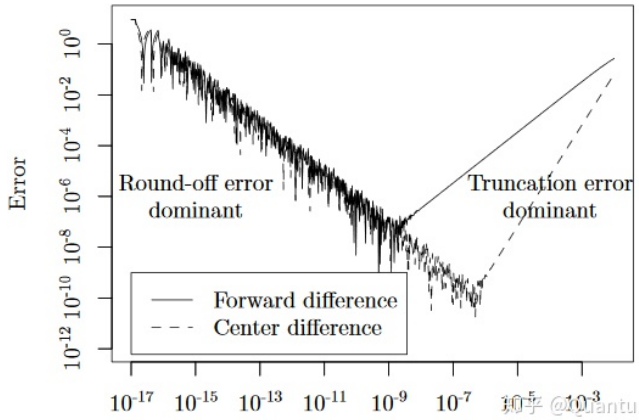


Manual or Symbolic Differentiations find the derivative of a given formula with respect to a specified variable through applying differential formulas, but they consume great computing power and too much time.

Table 1: Iterations of the logistic map $l_{n+1} = 4l_n(1 - l_n)$, $l_1 = x$ and the corresponding derivatives of l_n with respect to x, illustrating expression swell.

n	l_n	$\frac{d}{dx}l_n$	$\frac{d}{dx}l_n$ (Simplified form)
1	x	1	1
2	$4x(1-x)$	$4(1-x)-4x$	$4-8x$
3	$16x(1-x)(1-2x)^2$	$16(1-x)(1-2x)^2-16x(1-2x)^2-64x(1-x)(1-2x)$	$16(1-10x+24x^2-16x^3)$
4	$64x(1-x)(1-2x)^2(1-8x+8x^2)^2$	$128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$	$64(1-42x+504x^2-2640x^3+7040x^4-9984x^5+7168x^6-2048x^7)$

Numerical Differentiations find the numerical value of a derivative of a given function at a given point, but they can only find approximate solutions due to precision.



However, Automatic Differentiations could find the exact value of a derivative of a given function at a given point through breaking the complex formula into elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.), and by applying the chain rule repeatedly to these operations.

$$y = f(x_1, x_2) = \ln x_1 + x_1 * x_2$$

Compute $\frac{dy}{dx}$ at $(x_1, x_2) = (2, 5)$

Forward Primal Trace

$v_{-1} = x_1$		$= 2$
$v_0 = x_2$		$= 5$
$v_1 = \ln v_{-1}$		$= \ln 2$
$v_2 = v_{-1} \times v_0$		$= 2 \times 5$
$v_3 = \sin v_0$		$= \sin 5$
$v_4 = v_1 + v_2$		$= 0.693 + 10$
$v_5 = v_4 - v_3$		$= 10.693 + 0.959$
$y = v_5$		$= 11.652$

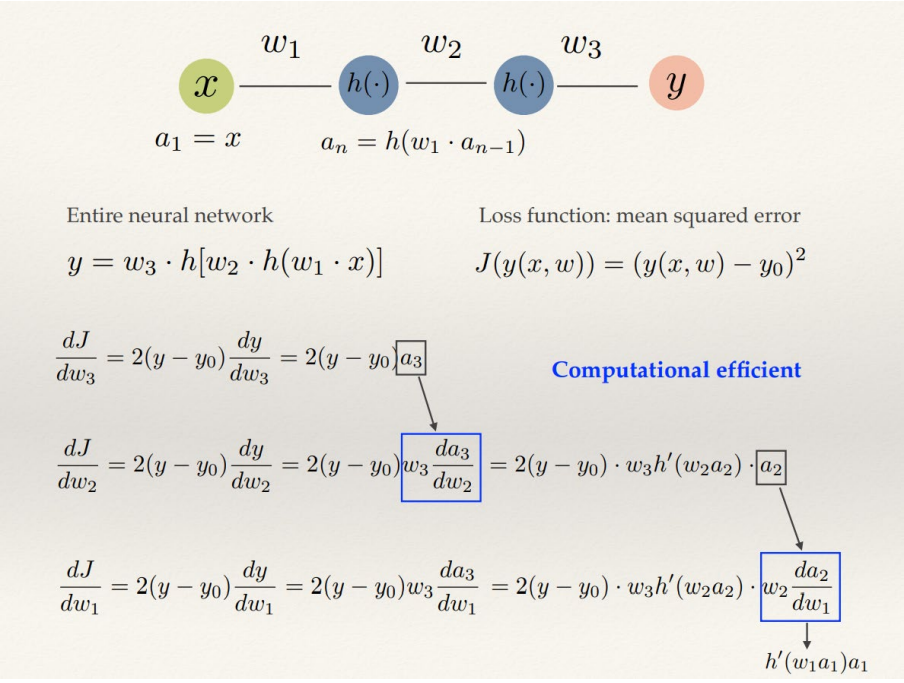
Forward Tangent (Derivative) Trace

$\dot{v}_{-1} = \dot{x}_1$		$= 1$
$\dot{v}_0 = \dot{x}_2$		$= 0$
$\dot{v}_1 = \dot{v}_{-1}/v_{-1}$		$= 1/2$
$\dot{v}_2 = \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1}$		$= 1 \times 5 + 0 \times 2$
$\dot{v}_3 = \dot{v}_0 \times \cos v_0$		$= 0 \times \cos 5$
$\dot{v}_4 = \dot{v}_1 + \dot{v}_2$		$= 0.5 + 5$
$\dot{v}_5 = \dot{v}_4 - \dot{v}_3$		$= 5.5 - 0$
$\dot{y} = \dot{v}_5$		$= \mathbf{5.5}$

Reverse Adjoint (Derivative) Trace

$\bar{x}_1 = \bar{v}_{-1}$			$= \mathbf{5.5}$
$\bar{x}_2 = \bar{v}_0$			$= \mathbf{1.716}$
$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$		$= \bar{v}_{-1} + \bar{v}_1 / v_{-1}$	$= 5.5$
$\bar{v}_0 = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$		$= \bar{v}_0 + \bar{v}_2 \times v_{-1}$	$= 1.716$
$\bar{v}_{-1} = \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$		$= \bar{v}_2 \times v_0$	$= 5$
$\bar{v}_0 = \bar{v}_3 \frac{\partial v_3}{\partial v_0}$		$= \bar{v}_3 \times \cos v_0$	$= -0.284$
$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2}$		$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1}$		$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3}$		$= \bar{v}_5 \times (-1)$	$= -1$
$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4}$		$= \bar{v}_5 \times 1$	$= 1$
$\bar{v}_5 = \bar{y}$		$= 1$	

In the expressions calculating the derivatives for each weight, they involve the parts from the previous layer. In this way, the Reverse Accumulation of Auto-Differentiation perfectly match the back propagation.

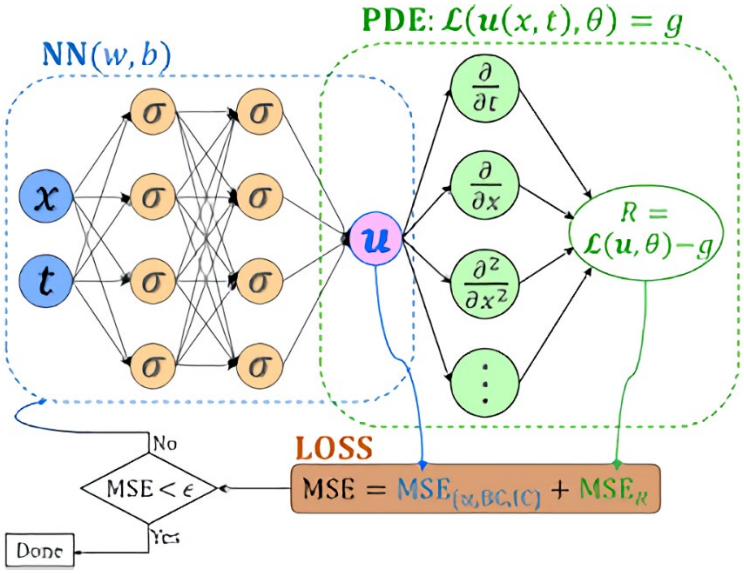


Thus, besides calculating the derivatives for weights, it could also calculate the derivatives between the output and inputs. Although these derivatives have little use for the traditional NNs, they are crucial to the Physical-informed Neural Network.

2.2 Physical-Informed Neural Networks

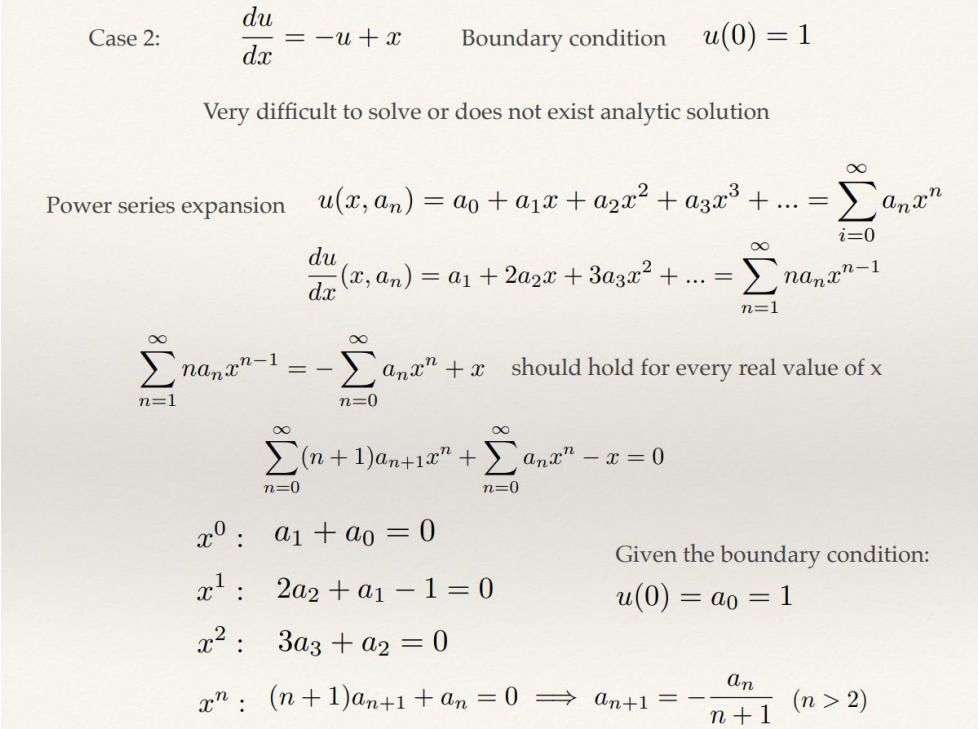
2.2.1 Differences with traditional Neural Networks

Physical-Informed Neural Networks (PINNs) utilize the vast amount of prior knowledge that is currently not being utilized by traditional machine learning. With this advantage, PINNs require much less data than NNs do. And the main difference between the NNs and PINNs is in their loss functions: NNs only use data to calculate loss, while PINNs involves the governing equations into it.



If we have differential equations that we need to solve, we only need finite boundary conditions to fit (one point, four lines, or six faces). Due to power series expansion, some complex functions can be approximated as simple polynomial functions. And because what we need to solve are equations, so left sides minus right sides are equal to zero. And

when x not equal to 0, after merging the same terms, we could get that the coefficient of each term equal to 0. And in this example, we only need one point to solve the whole equation.



Because of this characteristic, we can generate myriads of points within the range to help modify the model, since we only need to minimize the equation loss generated by these inputs to zero.

2.2.2 Coding

Initialization and Calculation

```
def __init__(self, t_x, u, x_f, layers, lb, ub, gamma):

    self.t_x = t_x
    self.u = u
    self.x_f = x_f

    self.lb = lb
```

```
        self.ub = ub

        self.layers = layers

        self.gamma = gamma

        # Initialize NNs
        self.weights, self.biases = self.initialize_NN(layers)

        # Create a list including all training variables
        self.train_variables = self.weights + self.biases
        # Key point: anything updates in train_variables will be
        #            automatically updated in the original tf.Variable

        # define the loss function
        self.loss = self.loss_NN()

        self.optimizer_Adam = tf.optimizers.Adam()
```

This is the `__init__` function, which takes seven parameters:

1. `t_x` is the set of points for the boundary conditions;
2. `u` is the set of values of the boundary conditions;
3. `x_f` is the set of points for calculating the equation loss;
4. `layers` is the configuration of the neural network, which is a list contains the number of neurons for each layer.
5. `lb` is the abbreviation for "lower bound," which is the value of the minimum value of the inputs;
6. `ub` is the abbreviation for "upper bound," which is the value of the maximum value of the inputs;
7. `gamma` is the coefficient of the equation loss.

This function calls `initialize_NN` and `loss_NN()`.

```
    def initialize_NN(self, layers):
        weights = []
        biases = []
        num_layers = len(layers)
        for l in range(0, num_layers - 1):
            W = self.xavier_init(size=[layers[l], layers[l + 1]])
            b = tf.Variable(tf.zeros([1, layers[l + 1]], dtype=tf.float32))
            weights.append(W)
            biases.append(b)
        return weights, biases

    def xavier_init(self, size):
        in_dim = size[0]
        out_dim = size[1]
        xavier_stddev = np.sqrt(2 / (in_dim + out_dim))
        return tf.Variable(tf.random.truncated_normal([in_dim, out_dim], stdde
v=xavier_stddev), dtype=tf.float32)
```

`initialize_NN` function is used to initialize the weights and biases for the neural networks. All biases are initialized to 0 and the weights are initialized through calling `xavier_init`. The values generated by this function follow a normal distribution with a specified mean and standard deviation, except that values whose mean is greater than 2 standard deviations are discarded and reselected.

```
    # calculate the physics-informed loss function
    def loss_NN(self):
        self.x_pred = self.net_u(self.t_x)
        loss_d = tf.reduce_mean(tf.square(self.u - self.x_pred))

        self.f_pred = self.net_f(self.x_f)
        loss_e = tf.reduce_mean(tf.square(self.f_pred))

        loss = loss_d + self.gamma * loss_e
        return loss
```

`loss_NN` function is used to calculate the overall loss of the training, which is data loss plus coefficient times equation loss. In order to get the data and equation loss, this function

calls `net_u` and `net_f`.

```
    def neural_net(self, X, weights, biases):
        num_layers = len(weights) + 1

        H = 2.0 * (X - self.lb) / (self.ub - self.lb) - 1.0
        for l in range(0, num_layers - 2):
            W = weights[l]
            b = biases[l]
            H = tf.tanh(tf.add(tf.matmul(H, W), b))
        W = weights[-1]
        b = biases[-1]
        Y = tf.add(tf.matmul(H, W), b)
        return Y

    '''
    Functions used to building the physics-informed contrainst and loss
    ===================================================================
    '''
    def net_u(self, t):
        res = self.neural_net(t, self.weights, self.biases)
        return res
```

`net_u` will call `neural_net` to process the Feed-Forward process. This line: `H = 2.0 * (X - self.lb) / (self.ub - self.lb) - 1.0` is used to normalize the input for better fitting. And the following lines process the matrixes addition, multiplication, and non-linearization during the training.

This is the end of the codes that can be shared between different PINN models. And this is not very different from the traditional neural network models. Next, we need to code different boundary conditions and formulas for different problems

2D Burgers' equation

Burgers' equation is a fundamental partial differential equation occurring in various areas of applied mathematics, such as fluid mechanics, nonlinear acoustics, gas dynamics, and traffic flow.

$$u_t + uu_x - (0.01/\pi)u_{xx} = 0, \quad x \in [-1, 1], \quad t \in [0, 1],$$

```
def net_f(self, xf):
    t = xf[:, 0:1]
    x = xf[:, -1:]
    f = self.govern(t, x, self.net_u)
    return f

def govern(self, t, x, func):
    with tf.GradientTape(persistent=True) as tape2:
        tape2.watch(t)
        tape2.watch(x)
        with tf.GradientTape(persistent=True) as tape:
            tape.watch(t)
            tape.watch(x)
            tx = tf.concat([t,x], 1)
            u = func(tx)
        u_t = tape.gradient(u, t)
        u_x = tape.gradient(u, x)
    u_xx = tape2.gradient(u_x, x)
    f = u_t + u * u_x - tf.cast((0.01 / math.pi), dtype='float32') * u_xx
    return f
```

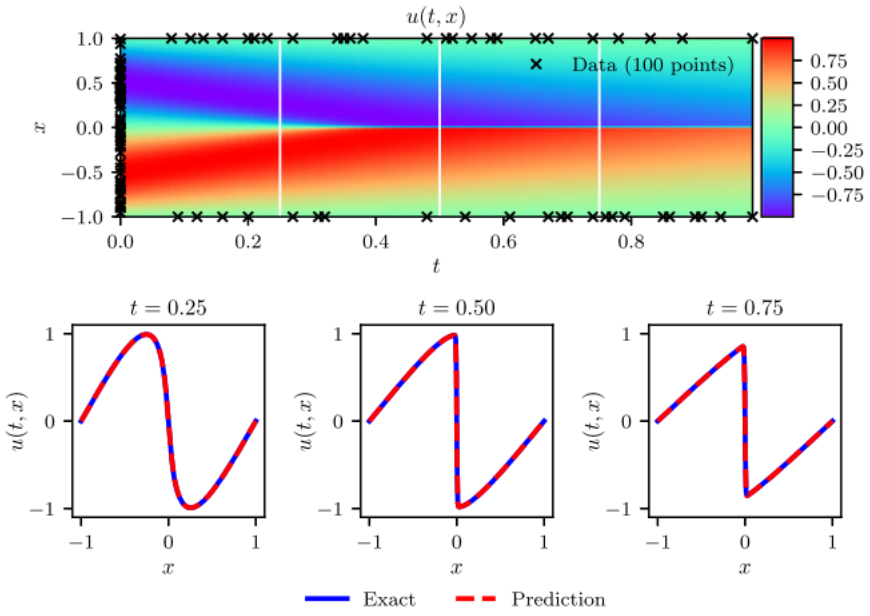
`net_f` will call `govern` to calculate the equation loss, which does the automatic differentiation. Since it has two variables and the equation involves second derivatives, we need two `tf.GradientTape` method and need to set `persistent=True` in order to watch two variables that the same time.

$$u(0, x) = -\sin(\pi x),$$
$$u(t, -1) = u(t, 1) = 0.$$

```
t = np.linspace(0, 1, 200, dtype='float32')[:, None]
x = np.linspace(-1, 1, 200, dtype='float32')[:, None]
T, X = np.meshgrid(t, x)

tx0 = np.hstack((T[0:1, :].T, X[0:1, :].T))
ux0 = np.zeros(x.shape, 'float32')

tx1 = np.hstack((T[-1:, :].T, X[-1:, :].T))
ux1 = np.zeros(x.shape, 'float32')

t0x = np.hstack((T[:, 0:1], X[:, 0:1]))
ut0 = -np.sin(math.pi * X[:, 0:1])

X_u_train = np.vstack([tx0, tx1, t0x])
u_train = np.vstack([ux0, ux1, ut0])

lb_t = t.min(0)[0]
ub_t = t.max(0)[0]
lb_x = x.min(0)[0]
ub_x = x.max(0)[0]

X_f_train = [lb_t, lb_x] + [ub_t - lb_t , ub_x - lb_x] * lhs(2, 3200)
X_f_train = np.vstack((X_f_train, X_u_train))

X_u_train = tf.cast(X_u_train, dtype=tf.float32)
u_train = tf.cast(u_train, dtype=tf.float32)
X_f_train = tf.cast(X_f_train,dtype=tf.float32)
```

Since the 2D Burgers' equation has three boundary conditions, so we have `tx0`, `tx1`, and `t0x`. `X_u_train` is the combination of the boundary conditions and `u_train` is the combinations of the values of them (`ux0`, `ux1`, and `ut0`). And `X_f_train` is the combination of boundary conditions and random points within the range for calculating the equation loss.

Here are the results from "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations"[2] 's Appendix A (Raissi, etc.):



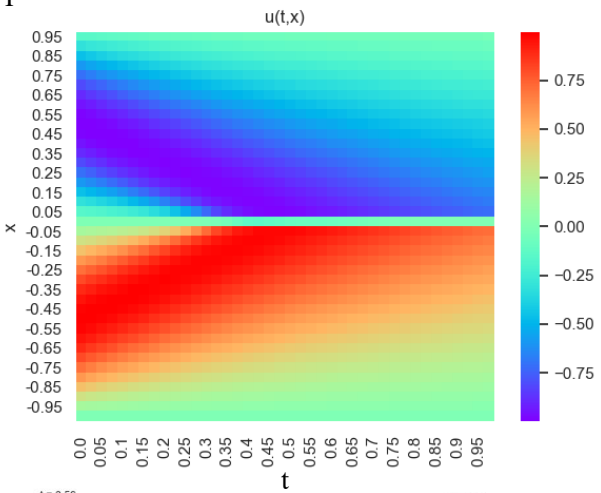
And here are the predictions from the previous codes:

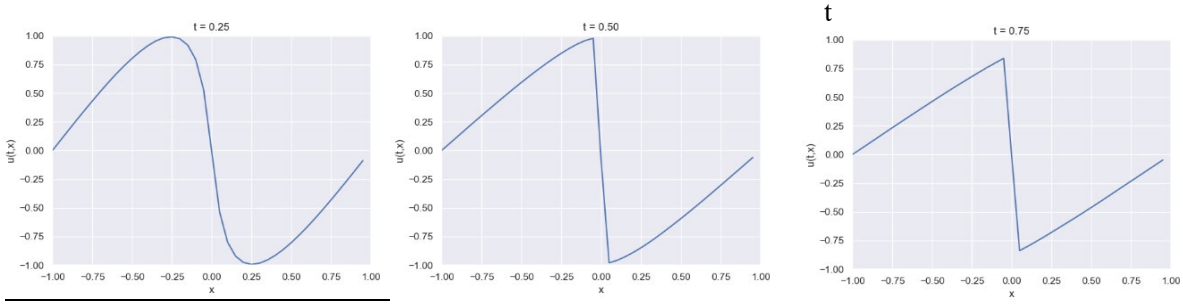
layers = [2, 50, 50, 50, 50, 1]

iteration = 50,000

gamma = 0.15[3]

Loss = 1.94e-06





[2] Raissi, M, etc. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations." *Journal of Computational Physics*. 2019. https://doi.org/10.1016/j.jcp.2018.10.045

[3] This value is obtained by dividing the equation loss after 500 iterations by the initial data loss.

Multi-variable partial differential equations

$$\frac{\partial}{\partial X}\left(4\frac{\partial U}{\partial X}+2\frac{\partial V}{\partial Y}\right)+\frac{\partial}{\partial Y}\left(\frac{\partial U}{\partial Y}+\frac{\partial V}{\partial X}\right)=\frac{\partial S}{\partial X}$$
$$\frac{\partial}{\partial Y}\left(4\frac{\partial V}{\partial Y}+2\frac{\partial U}{\partial X}\right)+\frac{\partial}{\partial X}\left(\frac{\partial U}{\partial Y}+\frac{\partial V}{\partial X}\right)=\frac{\partial S}{\partial Y}$$
$$S(X,Y)=-X^2-3Y^2$$

```
def govern(self, x, y, func):
    with tf.GradientTape(persistent=True) as tape2:
        tape2.watch(x)
        tape2.watch(y)
        with tf.GradientTape(persistent=True) as tape:
            tape.watch(x)
            tape.watch(y)
            t = tf.concat([x,y], 1)
            u, v = func(t)
        u_x = tape.gradient(u, x)
        u_y = tape.gradient(u, y)
        v_x = tape.gradient(v, x)
        v_y = tape.gradient(v, y)

    u_xx = tape2.gradient(u_x, x)
    u_xy = tape2.gradient(u_x, y)
    u_yy = tape2.gradient(u_y, y)

    v_xx = tape2.gradient(v_x, x)
    v_xy = tape2.gradient(v_x, y)
    v_yy = tape2.gradient(v_y, y)

    f = 4 * u_xx + 3 * v_xy + u_yy
    g = 4 * v_yy + 3 * u_xy + v_xx
    return f, g
```

Similar to 2D Burgers' equation, this project has two variables and involve second derivatives. But it contains two governing equations, so we need to return two equation loss.

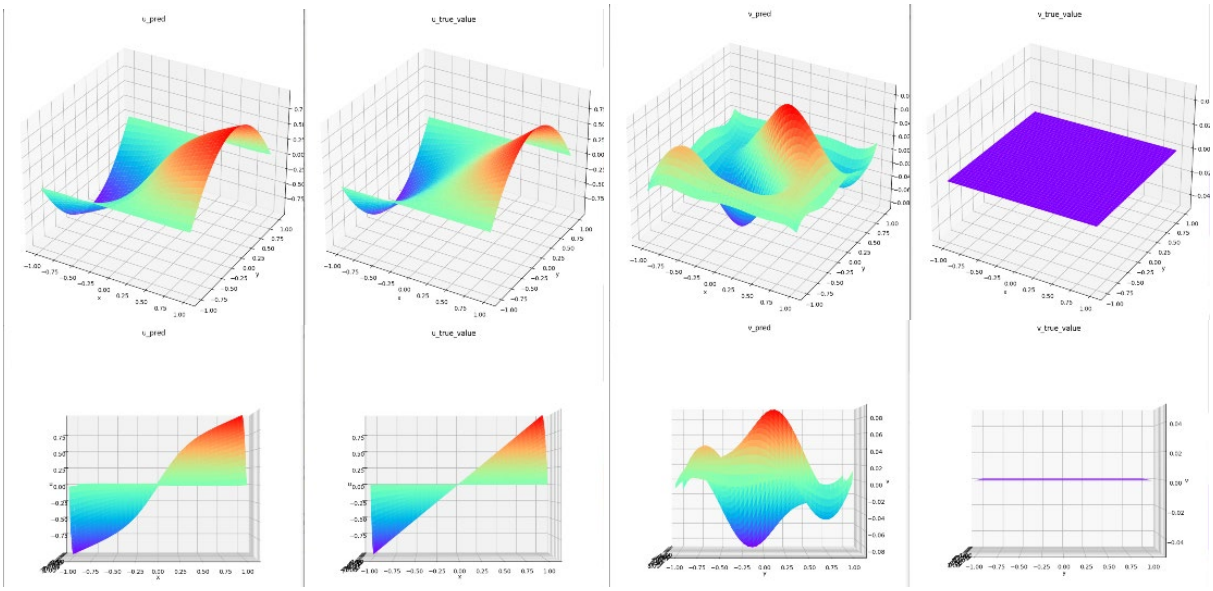
$$
\begin{aligned}
B.C. \quad & U(X=-1,Y) = -(1-Y^2), \\
& U(X=1,Y) = 1-Y^2, \\
& U(X,Y=\pm 1) = 0
\end{aligned}
$$

$$
B.C. \quad V(X=\pm 1,Y) = V(X,Y=\pm 1) = 0
$$

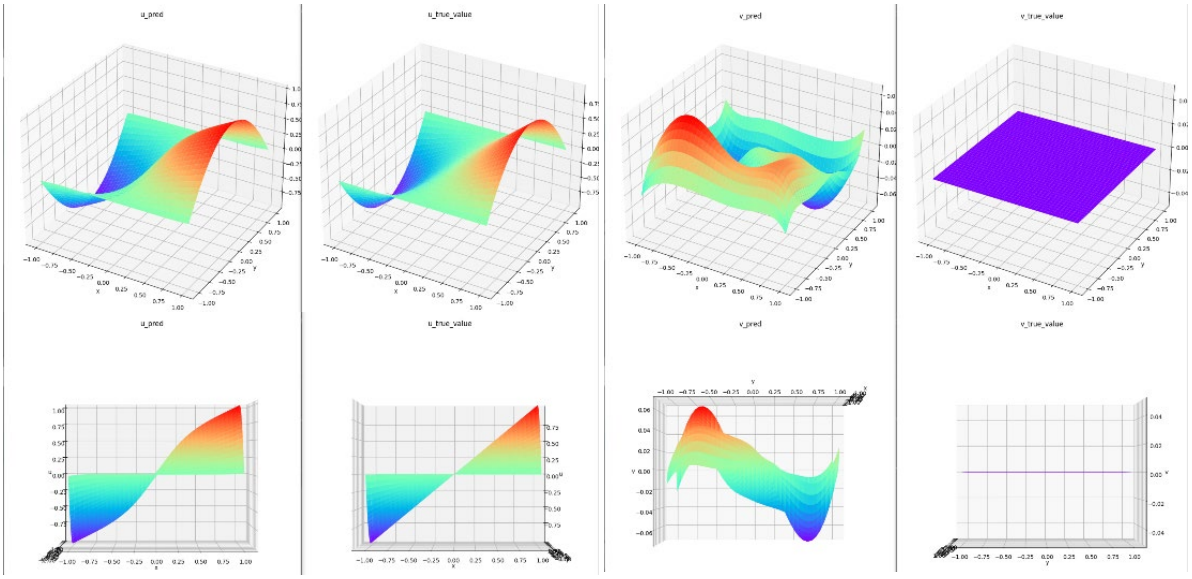
```
x = np.linspace(-1, 1, 200, dtype='float32')[:, None]
y = np.linspace(-1, 1, 200, dtype='float32')[:, None]
X, Y = np.meshgrid(x, y)

xy0 = np.hstack((X[0:1, :].T, Y[0:1, :].T))
uy0 = vy0 = np.zeros(y.shape, 'float32')

xy1 = np.hstack((X[-1:, :].T, Y[-1:, :].T))
uy1 = vy1 = np.zeros(y.shape, 'float32')

x0y = np.hstack((X[:, 0:1], Y[:, 0:1]))
ux0 = -(1 - Y[:, 0:1] ** 2)
vx0 = np.zeros(x.shape, 'float32')

x1y = np.hstack((X[:, -1:], Y[:, -1:]))
ux1 = 1 - Y[:, 0:1] ** 2
vx1 = np.zeros(x.shape, 'float32')

X_u_train = np.vstack([xy0, xy1, x0y, x1y])
u_train = np.vstack([uy0, uy1, ux0, ux1])
v_train = np.vstack([vy0, vy1, vx0, vx1])
u_v_train = np.hstack([u_train, v_train])

# Doman bounds
lb = x.min(0)
ub = x.max(0)

X_f_train = lb + (ub - lb) * lhs(2, 3200)
X_f_train = np.vstack((X_f_train, X_u_train))

X_u_train = tf.cast(X_u_train, dtype=tf.float32)
u_v_train = tf.cast(u_v_train, dtype=tf.float32)
X_f_train = tf.cast(X_f_train,dtype=tf.float32)
```

Both equations have four boundary conditions. However, except the values, their boundary conditions are the same: both x and y equal to one or minus one. And even for the values, when y equal to one or minus one, all the values equal to zero. So we have `xy0`, `xy1`, `x0y`, `x1y` and for boundary conditions and `uy0`, `uy1`, `ux0`, and `ux1` for u's boundary conditions values and `vy0`, `vy1`, `vx0`, `vx1` and for v's boundary conditions values. And `X_u_train`, `u_v_train`, and `X_f_train` are the same as those in the previous project.

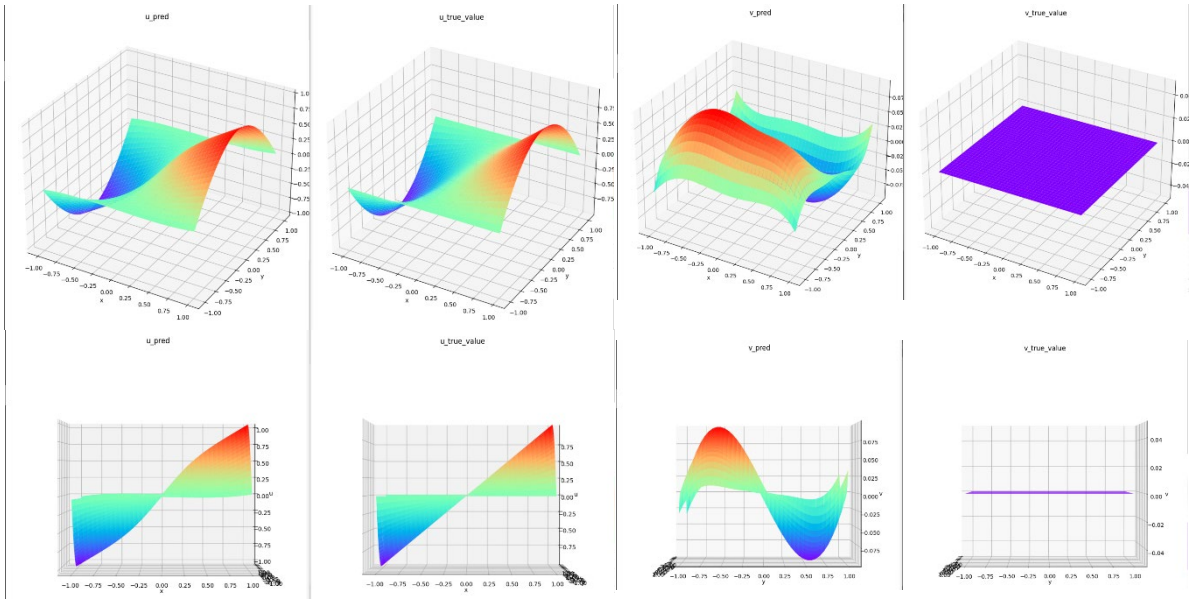
The exact solutions are $U(X,Y) = (1 - Y^2)X$ and $V(X,Y) = 0$. For this project I tried different gamma value to see the effects of it. The layer is [2, 50, 50, 50, 50, 2] and iteration is 50,000 for all the trainings.



gamma = 0.1, loss = 3.9e-5, u_error = 1.49e-1, v_error = 4.32e-2



gamma = 0.5, loss = 9.7e-5, u_error = 1.25e-1, v_error = 5.1e-2

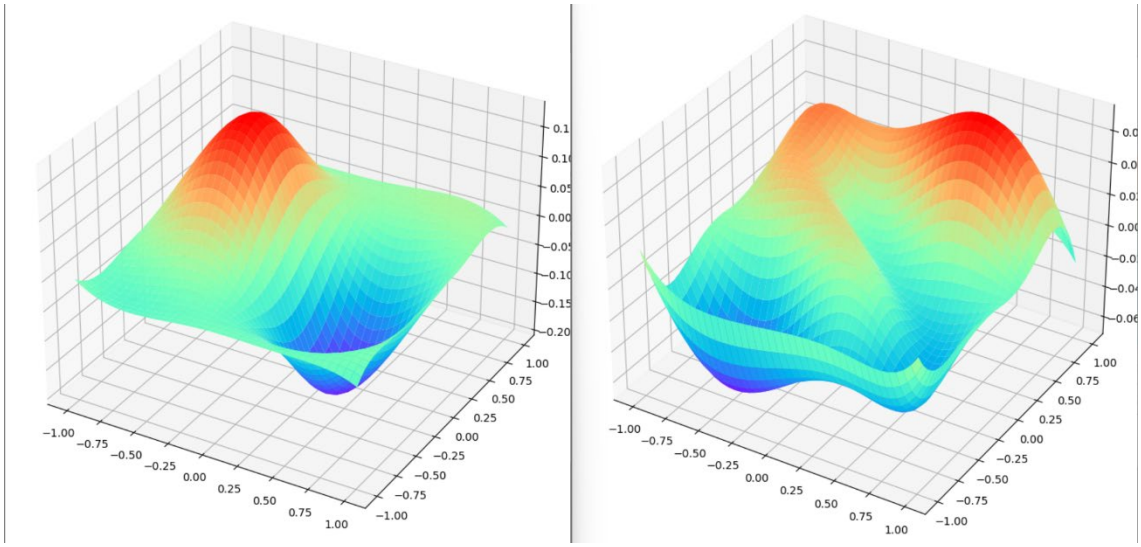


gamma = 1.0, loss = 2.94e-4, u_error = 7.65e-2, v_error = 9.0e-2

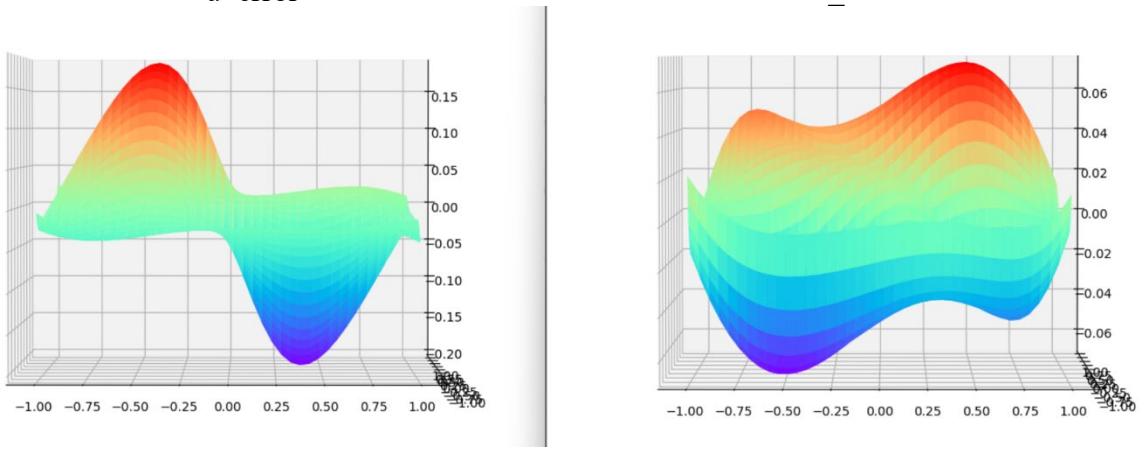
From the previous three trainings we could find that: as gamma increases, loss increases, u_error decreases but v_error increases. So, the larger the gamma, the better curve u is fitting and the worse cure v is fitting. The reason why these happen may be that both curves share

the most part of the neural networks and only separates at the last layer. So during the training, they interact with each other.

Also, although from the chart above, it seems that the variance of curve v is very large, but actually it is within 0.1 to -0.1. To show variance of prediction, we draw to error for each curve at gamma equal to 0.5:



u_error v_error



As we can see, the variance of u is much larger than that of v. The fitting to v is not as bad as it seems at the first glance.

3. Summary

Before this project, I had never been exposed to the knowledge of the direction of Artificial Neural Networks. And through this project, I have a deep understanding of the neural networks, especially how Physical-Informed Neural Networks work, like the matrix multiplication, loss function, gradient descent, automatic differentiation, and understand their physical meanings, like how to fit high-frequency functions, and so on. In this process, Teacher explained the working principle of neural networks to me carefully, patiently answered my questions, and gave me programming homework to deepen my understanding of the operation processes of neural networks. All these helped me gain a deep understanding.

This project improved my programming ability and gave me a deep and comprehensive understanding of the field of neural networks. I will make good use of the knowledge and ability I have learned from this project to further explore this field and pave the way for the future development in the field of artificial intelligence.