Guide to Tensorflow

Jiajing Guan

December 25, 2020

1 Installation

If you don't want to use Tensorflow on GPUs, the installation process is fairly simple. You can run the following command in the command line to install Tensorflow:

```
pip install tensorflow
```

You can use functions of Tensorflow by importing Tensorflow in the preamble section of the Python script:

```
import tensorflow as tf
```

2 Basic Concepts in Tensorflow

Tensorflow is very similar to Numpy. They both have data structures defined by the package. For Tensorflow, they have their own versions of constants and variables, which can be processed faster than primitive Python structures. But some Tensorflow functions could take Numpy arrays as inputs. It is always helpful to refer to the Tensorflow function documentation before using it. For example, if we want to use an array of numbers as a Tensorflow constant, we would use:

```
A = tf.constant([1,2,3,4],dtype=tf.float32)
```

If we want to allocate space for a variable that we will update over and over again, like the weights in the network, we would use the following line:

```
B = tf.Variable(np.zeros([32,32]),trainable=True,dtype=tf.float32)
```

Notice that there are many attributes to Tensorflow variables. The `trainable` attribute will influence how Tensorflow track the variable throughout operations. If we want to update the variable, there are dedicated functions, like `B.assign()`, in Tensorflow that will perform the task.

Now we will use the `NN_tf.py` script to explore more functionality of Tensorflow.

3 Neural Networks

Let's consider what functions we would need in a neural network class. We will need functions that initialize weights, perform forward passes, compute the loss, take the gradient and Jacobian matrix of the loss with respect to weights and update weights. There are more aspects to the neural network class, but we will focus on three areas: initializing weights, forward pass and computing loss, computing gradients and Jacobian matrix.

Before going into the details of each part, we need to consider the necessary input to initialize a neural network class. We need to define the dimension of input and output, dimension of the hidden layers and the environment instance of the PDE problem. So we will setup the neural network class as:

```
class NN_tf:
        def __init__(self, input_size, output_size, layers, env):
                self.env = env
                self.name = "DNN"
                self.input_size = input_size
                self.output_size = output_size
                self.layers = layers
                self.lb = tf.constant(self.env.lb, dtype = tf.float32)
                self.ub = tf.constant(self.env.ub, dtype = tf.float32)
                self.initialize_NN()
```

The `self.lb` and `self.ub` are lower and upper bounds of the input. We will use these variables later in the forward pass to normalize the input. The `self.initialize_NN()` is the function that initialize the weights.

3.1 Initialize Weights

Now we will talk about `self.initialize_NN()` function. In my implementation, I put all the weights in one list and all the biases in another list. There are other ways to store these weights. In fact, Tensorflow has an almost out-of-the-box implementation for networks called `model`. I didn't want to use their implementation as it is harder to control the shape and initialization of the weights. But a useful link explaining how to use `model` is here.

To initialize the weights, we need to decide the random generation scheme to generate the entries in the weights. I used Xavier initialization for weights and used zeros for biases. Xavier initialization sets a layer's weights to values chosen from a random uniform distribution that's bounded between $\pm\frac{\sqrt{6}}{\sqrt{n_i, n_{i+1}}}$, where n_i is the number of incoming network connections, and n_{i+1} is the number of outgoing network connections from that layer. Thus, we can create a function that takes tuple (n_i, n_{i+1}) as an input, yields a Tensorflow variable with values generated by the Xavier initialization:

```
def xavier_init(self, size, name):
        xavier_stddev = np.sqrt(6)/np.sqrt(np.sum(size))
        return tf.Variable(tf.random.normal(size, mean = 0,\
         stddev = xavier_stddev), dtype=tf.float32, name = name)
```

The name variable is used to assign a name to the Tensorflow variable. It is not necessary but the name is helpful when we are debugging the code.

In terms of the shape of the weights, I flattened all the weights and biases to a row vector so that later when we take the Jacobian matrix, we don't need to deal with reshaping issues. But this does mean that we need to reshape the weights during the forward pass. To do that, we will create list to store the shapes of weights and biases and reshape the weights during the forward pass. In the input layer, the weight should be of size $D_{input} \times D_{hidden}$; in the output layer, the weight should be of size $D_{hidden} \times D_{output}$. The size of weights and biases in the hidden layers will be determined by the width of the hidden layer. If we set the width to be constant throughout the hidden layers, the size of the weight would be $D_{hidden} \times D_{hidden}$. Refer to `initialize_NN` function in the `NN_tf.py` script for implementation. There is one thing to keep in mind. The length of the `self.layers` variable should be 1 larger than the number of hidden layers. This has to do with an implementation issue.

For example, we want to create a network with 2 hidden layers of width 32 to use PINN to solve the 1D convection-diffusion problem. The problem determines that the input size of the network would be 2 (dimension of x coordinates plus dimension of ξ) and the output size would be 1 (dimension of u). The `layers` input should be `[32,32,32]`.

3.2 Forward Pass and Loss Function

A forward pass in a network can be represented mathematically by the following equations:

$$\begin{aligned} Y_0 &= X_{in} \\ Y_{n+1} &= \sigma(W_n Y_n + b_n) \\ Y_{N+1} &= W_{N+1} Y_N + b_{N+1} \end{aligned} \tag{1}$$

where $X_0 = X_{in}$ are the sample inputs, $\sigma(\cdot)$ is a nonlinear function referred to as the activation function and $\boldsymbol{\theta}$ denotes the collection of weights W_j and biases b_j, for $j = 1, \ldots, N+1$. Here, the integer N will represent the number of hidden layers.

To make the neural network class as general as possible, we set the forward pass function to take x, y, t and ξ Tensorflow constants as input. The implementation is shown below:

```
def forward(self, x_tf, y_tf, t_tf, xi_tf):
        num_layers = len(self.weights)
        X = tf.concat((x_tf, y_tf, t_tf, xi_tf), axis = -1)
        H = (X - self.lb)/(self.ub - self.lb)
        for l in range(num_layers-1):
                W = tf.reshape(self.weights[l], self.weights_dims[l])
                b = self.biases[l]
                H = tf.keras.activations.tanh(tf.matmul(H, W) + b)
        W = tf.reshape(self.weights[-1], self.weights_dims[-1])
        b = self.biases[-1]
        H = tf.matmul(H, W) + b
        return H
```

The process of the implementation can be summarized as concatenating the input variables, normalizing the input and performing the operations in each layer. Notice that the activation function is tanh here, indicated by the `tf.keras.activations.tanh` function. There are other out-of-the-package activation function provided by Tensorflow. You can find out the available functions here.

But it is often true that not all inputs are needed in a PDE problem. In those cases, we initialize the unused inputs as empty Tensorflow constants. Luckily, Tensorflow allows empty arrays to have shapes and that permits us to concatenate empty variables without issues.

Now we discuss how we compute the loss. To perform PINN, we need the Mean Squared Error (MSE) of different types of samples. Thus, I chose to organize the samples as a list of dictionaries. Each dictionary will contain the necessary inputs to perform a forward pass, target samples and information such as number of samples N, weighting for this type of samples, and the type of sample ("Res" for residual, "B_D" for Dirichlet boundary conditions, "B_N" for Neumann boundary conditions and "Init" for initial condition or true solution). The loss function will add the MSE of each type of samples. The implementation of the loss function is shown below:

```
def loss(self, samples_list):
        loss_val = tf.constant(0, dtype = tf.float32)
        for i in range(len(samples_list)):
                dict_i = samples_list[i]
                name_i = dict_i["type"]
                x_tf = dict_i["x_tf"]
                y_tf = dict_i["y_tf"]
                t_tf = dict_i["t_tf"]
                xi_tf = dict_i["xi_tf"]
                target = dict_i["target"]
                N = dict_i["N"]
```

```
            weight = dict_i["weight"]

            if name_i == "Res":
                    f_res = self.compute_residual(x_tf, y_tf, t_tf, xi_tf, target)
                    f_u = f_res*np.sqrt(weight/N)
                    loss_f = tf.math.reduce_sum(f_u**2)/2
                    loss_val = loss_val + loss_f

            elif name_i == "B_D":
                    err_do = self.compute_solution(x_tf, y_tf, t_tf, xi_tf, target)
                    err_d = err_do*np.sqrt(weight/N)
                    loss_d = tf.math.reduce_sum(err_d**2)/2
                    loss_val = loss_val + loss_d

            elif name_i == "B_N":
                    err_n = self.compute_neumann(x_tf, y_tf, t_tf, xi_tf,target)
                    err_n = (err_n)*np.sqrt(weight/N)
                    loss_n = tf.math.reduce_sum(err_n**2)/2
                    loss_val = loss_val + loss_n

            elif name_i == "Init":
                    err_0 = self.compute_solution(x_tf, y_tf, t_tf, xi_tf, target)
                    err_0 = (err_0)*np.sqrt(weight/N)
                    loss_0 = tf.math.reduce_sum(err_0**2)/2
                    loss_val = loss_val + loss_0

    return loss_val
```

For each each of samples, the inputs will be passed to a separate function that will compute the errors compared to the targets. The simplest one to understand is probably the `compute_solution` function, which is used in computing errors for Dirichlet boundary and initial condition samples. This function simply computes the difference between the network output and the target. The implementation is shown below:

```
def compute_solution(self, x_tf, y_tf, t_tf, xi_tf, target):
        u_p = self.forward(x_tf, y_tf, t_tf, xi_tf)
        err = u_p - target
        return err
```

To compute the errors for residual and Neumann boundary condition, we need to first take derivative of the network output with respect to the input, then computes the residual value or the Neumann boundary condition based on the specific problem. We will talk about how we can take derivatives of the output with respect to the input in the next section. For now, let's assume that we have computed quantities u_x, u_y, u_t, u_{xx} and u_{yy}. We will now provide these quantities to the residual function and Neumann boundary condition function defined in `self.env` class. After obtaining the outputs from residual function and Neumann boundary condition, we compute the difference between the output and the sample target. The implementations of function `compute_residual` and `compute_neumann` are shown below:

```
@tf.function
def compute_residual(self, x_tf, y_tf, t_tf, xi_tf, target):
        u, u_x, u_y, u_t, u_xx, u_yy = self.derivatives(x_tf, y_tf, t_tf, xi_tf)
        f_res = self.env.f_res(x_tf, y_tf, t_tf, xi_tf, u, u_x, u_y, u_t, u_xx, u_yy)
        f_err = f_res - target
        return f_err

@tf.function
def compute_neumann(self, x_tf, y_tf, t_tf, xi_tf, target):
```

```
        u, u_x, u_y, u_t, u_xx, u_yy = self.derivatives(x_tf, y_tf, t_tf, xi_tf)
        ub_n_p = self.env.neumann_bc(u_x, u_y)
        err = ub_n_p - target
        return err
```

So how are the residual function and the Neumann boundary condition function defined? They are quite simple. With the given quantities, we can manipulate them according to the nonlinear differential operator or the Neumann boundary condition defined in the problem. For example, in the 1D convection-diffusion problem, the residual function is defined as follow:

```
def f_res(self, x_tf, y_tf, t_tf, xi_tf, u, u_x, u_y, u_t, u_xx, u_yy):
        f_u = -u_xx*xi_tf+u_x
        return f_u
```

Since the 1D convection-diffusion problem does not have a Neumann boundary condition, it will just return nothing:

```
def neumann_bc(self, u_x, u_y):
        return
```

3.3 Gradient and Jacobian Matrix

In Tensorflow, they have an auto-differentiation capability that will track how trainable variables are used. To use this functionality, we need to use `GradientTape` that is provided by Tensorflow. There are lots of examples on using `GradientTape`. The tutorial provided by Tensorflow is really helpful.

For PINN, we need to first take derivative of the output with respect to some inputs to obtain the desired quantities in the nonlinear differential operator. We then need to take gradient or Jacobian matrix of the error with respect to the weights. We will achieve both tasks using the `GraidentTape`. Let's first focus on taking derivatives of outputs with respect to inputs. Since we often need second derivatives, we need a nested `GradientTape` structure. The idea is to perform a forward pass within the inner `GradientTape` while asking the `GradientTape` to watch how the inputs are being used in the operations. Once a forward pass is performed, we can take first derivatives of outputs with respect to inputs using the `gradient` function with the inner `GradientTape`. We then take first derivatives with respect to the inputs using the `gradient` function again to obtain second derivatives. The implementation is shown below:

```
def derivatives(self, x_tf, y_tf, t_tf, xi_tf):
        with tf.GradientTape(persistent = True) as tape1:
                tape1.watch(x_tf)
                tape1.watch(y_tf)
                tape1.watch(t_tf)
                with tf.GradientTape(persistent = True) as tape:
                        tape.watch(x_tf)
                        tape.watch(y_tf)
                        tape.watch(t_tf)
                        H = self.forward(x_tf, y_tf, t_tf, xi_tf)
                u_x = tape.gradient(H, x_tf)
                u_y = tape.gradient(H, y_tf)
                u_t = tape.gradient(H, t_tf)
        u_xx = tape1.gradient(u_x, x_tf)
        u_yy = tape1.gradient(u_y, y_tf)
        return H, u_x, u_y, u_t, u_xx, u_yy
```

We have discussed the loss function in the previous section. Now we want to take the loss value with respect to the weights. The process is similar to before. We set up a `GradientTape` and obtain the loss value within the tape. Then we take gradient of the loss value with respect to the list of weights and biases. It is often that the weight and bias in the output layer do not get used in the loss computation (because the

residual function often doesn't use u, but uses first and second derivatives). In those cases, we would obtain a `None` for the gradient. We then replace those `None` with zeros of appropriate sizes. The implementation is shown below:

```
def construct_Gradient(self, samples_list):
        with tf.GradientTape(persistent = True) as tape:
                loss_val = self.loss(samples_list)
        weights_grads = tape.gradient(loss_val, self.weights)
        biases_grads = tape.gradient(loss_val, self.biases)
        if biases_grads[-1] is None:
                biases_grads[-1] = tf.zeros([1, self.output_size])
        grads_tol_W = tf.transpose(tf.concat(weights_grads, axis = -1))
        grads_tol_b = tf.transpose(tf.concat(biases_grads, axis = -1))
        return loss_val, grads_tol_W, grads_tol_b
```

Now we talk about how to take the Jacobian matrix. In my implementation, I set up separate functions to take Jacobian matrices for different type of samples. This is because each type of samples require different procedures to compute the errors. But these functions that take Jacobian matrices are similar. The only variation between them is the function used to compute error. Thus, we will just use the function that computes the Jacobian matrix for errors of residual points as a template. To obtain the Jacobian matrix, we set up a `GradientTape` and compute the error within the tape. Then we use the `jacobian` function to take the Jacobian matrix. Like the case for gradient, we replace `None` with zeros of appropriate sizes. The implementation is shown below:

```
def construct_Jacobian_residual(self, x_tf, y_tf, t_tf, xi_tf, target, N, weight):
        with tf.GradientTape(persistent = True) as tape:
                f_res = self.compute_residual(x_tf, y_tf, t_tf, xi_tf, target)
                err = (f_res)*tf.math.sqrt(weight/N)
                err = tf.reshape(err, [tf.reduce_prod(tf.shape(err)),1])
        weights_jacobians = tape.jacobian(err, self.weights)
        biases_jacobians = tape.jacobian(err, self.biases)
        if biases_jacobians[-1] is None:
                biases_jacobians[-1] = tf.zeros([tf.shape(biases_jacobians[0])[0], \
                        self.output_size, 1, self.output_size])
        jacobs_tol_W = tf.squeeze(tf.concat(weights_jacobians, axis = -1))
        jacobs_tol_b = tf.squeeze(tf.concat(biases_jacobians, axis = -1))
        del tape
        return jacobs_tol_W, jacobs_tol_b, err
```

4 Tensorflow Graph

In the `NN_tf.py` file, you will see that a lot of functions have a `@tf.function` as the prefix. This declaration tells Tensorflow to compile the function and track the operations to create a graph on the function. Once a graph is created, if the same type of inputs are provided, Tensorflow will employ the previously constructed graph and the operations will become a streamline. This is extremely helpful because during the operation process, we need to take gradients and Jacobian matrices over and over again. The computational cost was significantly reduced ever since I used the Tensorflow graph.

However, there is a caveat to `@tf.function`. Tensorflow will construct a new graph every time a different type of input is provided, which is time consuming. Also, Tensorflow has a hard time recognizing primitive Python data types. Thus, it is recommended to use Tensorflow constants or variables as inputs and outputs if one wants to use `@tf.function`.