Session 2 part 2 - Introduction to Tensorflow

Jamie M. Taylor (CUNEF Universidad, Madrid, Spain)
jamie.taylor@cunef.edu

BCAM - In-Deep workshop

The code

- Most of the code I'm going to present in these slides is all available on github,

`https://github.com/jamie-m-taylor/In-Deep-examples`

- You may find it helpful to follow the code as I discuss it.
- Because of the size of the slides, many comments etc are not here, but are available on the git.

- Tensorflow is a large library for tools used for solving problems arising in machine learning.
- It is full of "black-box" functionality - one can very quickly and easily define a neural network and implement many standard problems coming from data analysis.
- Tensorflow is not designed for PDEs - we will have to rip open the black box to make it do what we want.
- The first thing we need to do is install it, then we load it with

```
import tensorflow as tf
```

- Much like Python itself, some fundamental objects are integers (e.g. `int32`), floats (e.g. `float32`) and Booleans (`bool`).
- In our applications, we will generally be considering manipulating large tensors constructed from these data types.
- In tensorflow, these can be defined as either `constants` or `Variables`. They are defined similarly to numpy arrays, and must have an $N_1 \times N_2 \times N_3 \ldots$ structure. E.g., they cannot be "triangular" arrays.
- We can define them with commands such as

```
constant_tensor  = tf.constant([1.,2.,3.,4.])
variable_tensor = tf.Variable([[1.,2.],[3.,4.]])
float_64_tensor = tf.constant([[3.,4.],[5.,3.],[1.,2.]],
                              dtype="float64")
```

- Tensors have shapes that can be called by `tf.shape`, e.g.

```
tf.shape(tf.constant([1.,2.,3.,4.]))
### <tf.Tensor:shape = (1,),dtype=int32, numpy=array([4])
```

- When tensors have the *same* shape, addition and multiplication work componentwise

```
x1 = tf.constant([[1.,2.],[-1.,-2.]])
x2 = tf.constant([[3.,4.],[5.,6.]])

x1+x2
# <tf.Tensor: shape=(2, 2), dtype=float32, numpy=
# array([[4., 6.],
#        [4., 4.]], dtype=float32)>

x1*x2
# <tf.Tensor: shape=(2, 2), dtype=float32, numpy=
# array([[  3.,   8.],
#        [ -5., -12.]], dtype=float32)>
```

- When adding/multiply tensors with different shapes, it will try to "broadcast", by extending one by repetition to make it the same shape as the other.

```
x1 = tf.constant([[1.,2.],[5.,6.]])
x2 = tf.constant([[0.],[1.]])
x3 = tf.constant([[0.,0.],[1.,1.]])

x1+x2
#<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
#array([[1., 2.],
#       [6., 7.]], dtype=float32)>

x1+x3
#<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
#array([[1., 2.],
#       [6., 7.]], dtype=float32)>
```

- All of your favourite calculus functions are available with `tf.math` and act componentwise. For example,

```
x1 = tf.constant([[1.,2.],[5.,6.]])

tf.math.exp(x1)
#<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
#array([[  2.7182817,   7.389056 ],
#       [148.41316  , 403.4288   ]], dtype=float32)>

tf.math.sqrt(x1)
#<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
#array([[1.       , 1.4142135],
#       [2.236068 , 2.4494898]], dtype=float32)>
```

Some tensor-based operations are the following.

- `tf.reduce_sum` - This computes the sum over all elements in the tensor (scalar output)
- `tf.reduce_mean` - The arithmetic mean over all elements.
- `tf.random.uniform(shape,minval=x1,maxval=x2)` - A random, uniform sample of shape `shape`, with minimum and maximum values `x1` and `x2`
- `tf.random.normal(shape,mean=mu,stddev=sigma)` - A random sample from a normal distribution.

Autodiff

- When taking gradients of a loss function or evaluating derivatives inside the loss itself, we need to take derivatives.
- Tensorflow can find the derivative of (almost) anything it can evaluate via *autodiff*.
- There are two modes, forward and backward, and I will focus on backward mode in this course.
- One must use a `GradientTape` to "record" the calculations - this allows the backpropagation to calculate the derivative - backward autodiff is based on a "forward pass" to evaluate the quantity of interest, then a "backward pass" through the calculations to evaluate the derivatives.
- Variables are automatically "watched" by tensorflow, whilst you need to tell tensorflow to watch constants.

Autodiff - Gradients

```
x = tf.constant([1.,2.])

with tf.GradientTape() as t1:  #Start recording the calculations
    t1.watch(x)                #Tell it to watch x
    z = tf.reduce_sum(x**2)    #Evaluate z=x[0]^2+x[1]^2
dz = t1.gradient(z,x)          #Find the gradient of z wrt x

#<tf.Tensor: shape=(2,), dtype=float32, numpy=array([2., 4.], dtype=
    float32)>
```

- As `x` is a constant, if I don't include `t1.watch(x)`, `dz` gives a `None` value.
- If `x` is a Variable, I don't need to include `t1.watch(x)`.

Autodiff

- Higher order gradients can be done in the same way - we use several layers of gradient tapes.

```
x = tf.Variable([4.])

with tf.GradientTape() as t2:       #First GradientTape
    with tf.GradientTape() as t1:   #Second GradientTape
        z = x**2                    #Evaluate z=x**2
    dz = t1.gradient(z,x)           #Find the gradient of z wrt x
d2z = t2.gradient(dz,x)             #Find the gradient of dz wrt x
#<tf.Tensor: shape=(1,), dtype=float32, numpy=array([2.], dtype=
    float32)>
```

Autodiff

- If we have a vector-valued function with a vector input, we may want its Jacobian.
- The Hessian is a simple example, which is the Jacobian of the gradient.

```
x = tf.Variable([2.,3.])

with tf.GradientTape() as t2:       #First GradientTape
    with tf.GradientTape() as t1:   #Second GradientTape
        z = tf.reduce_sum(x**2)     #Evaluate z=x**2
    dz = t1.gradient(z,x)           #Find the gradient of z wrt x
Hz = t2.jacobian(dz,x)              #Find the Jacobian of dz wrt x
#<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
#array([[2., 0.],
#       [0., 2.]], dtype=float32)>
```

- When evaluating a loss function based on an ODE, we want to evaluate derivatives of a function at each point. That is, we have N data points (x_n) and N outputs $u(x_n)$. By taking X to be a size N tensor, $u(X)$ is a size N tensor, and `gradient` acts componentwise

```
x = tf.Variable([1.,2.,3.])

with tf.GradientTape() as t1:
    u= x**2                         #u is the square of x componentwise
du = t1.gradient(u,x)               #obtain the componentwise derivatives
#<tf.Tensor: shape=(3,), dtype=float32, numpy=array([2., 4., 6.],
    dtype=float32)>
```

- The same can be done if u has vectorial input. For example,

```
xy = tf.Variable([[1.,2.],[2.,3.],[4.,5.]]) ##Pairs (x,y).

with tf.GradientTape() as t1:
    u= tf.reduce_sum(xy**2,axis=-1)    ##Evaluates x^2+y^2 at each
        pair
    #u=<tf.Tensor: shape=(3,), dtype=float32, numpy=array([ 5., 13.,
        41.], dtype=float32)>
du = t1.gradient(u,xy)
#<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
#array([[ 2.,  4.],
#       [ 4.,  6.],
#       [ 8., 10.]], dtype=float32)>
```

- Alternatively, you can take separate tensors for `x` and `y`, and take the derivative with each.
- Note, `du` is now a list of tensors, `[dux,duy]`.
- Depending on what you wish to do, this may be more convenient.

```
x = tf.Variable([1.,2.,4.])
y = tf.Variable([2.,3.,5.])

with tf.GradientTape() as t1:
    u= x**2+y**2
du = t1.gradient(u,[x,y])
#[<tf.Tensor: shape=(3,), dtype=float32, numpy=array([2., 4., 8.],
    dtype=float32)>,
# <tf.Tensor: shape=(3,), dtype=float32, numpy=array([ 4.,  6.,
   10.], dtype=float32)>]
```

- `GradientTape` by default "forgets" everything after the first gradient is taken, if you want to take several gradients, you must use `persistent=True`.

```
x1 = tf.Variable([1.,2.,3.])

with tf.GradientTape(persistent=True) as t1:

    z1 = tf.reduce_sum(x1**2)
    z2 = tf.reduce_sum(x1**3)

dz1 = t1.gradient(z1,x1)
dz2 = t1.gradient(z2,x1)

del t1
```

- `del` tells the `GradientTape` to stop monitoring (frees up resources)

Graph mode

- Tensorflow can run in two modes, *Eager mode* or *Graph mode*.
- The precise details of Graph mode are complex, but the idea is that it creates a highly optimised version of your function.
- It also offers other flexibility, not requiring Python itself, but our priority today is speed.
- Defining a function to run in graph mode is simple. Just prefix the definition of your function with `@tf.function`

```
def square_eager(x):    #A function that runs in Eager mode when called
        return x**2

@tf.function
def square_graph(x) #A function that runs in Graph mode when called
        return x**2
```

- Let's see how fast it runs in comparison.

```
def deriv_sin(x):   #Function to derive sine in Eager mode
    with tf.GradientTape() as t1:
        t1.watch(x)
        s = tf.math.sin(x)
    ds = t1.gradient(s,x)
    return ds

@tf.function
def deriv_sin_graph(x):  #Function to derive sine in Graph mode
    with tf.GradientTape() as t1:
        t1.watch(x)
        s = tf.math.sin(x)
    ds = t1.gradient(s,x)
    return ds
```

- I take `x` to be a random tensor with 10^8 entries.

```
Time (Eager) 0.33080077171325684
Time (Graph, first call) 0.21370434761047363
Time (Graph, second call) 0.07053089141845703
```

- I run `deriv_sin` once, and `deriv_sin_graph` twice.
- The graph mode is must faster each time.
- The graph mode takes three times longer on the first call than the second.
- This is because it needs to build the graph, and this has its own cost.
- Once constructed, it is significantly faster (almost 5 times)

- When using Tensorflow's training loops, it automatically puts things into graph mode - you don't need to do anything.
- Warning: If you change the shape or type of inputs, the graph needs to be reconstructed, and this has a cost.
- Changing the *values* does not require a new graph, as long as the shape stays the same.
- This is why it is powerful for iterative methods - you do the same calculations over and over with the same type and shape of data, and it is optimised to do this.
- Once trained, this also means the model can make very fast predictions on a given type of data.

Defining a simple architecture

- Tensorflow can build simple NNs for you. We will consider a fully-connected feedforward neural network. These are made of `Dense` layers. We can define a simple NN with a single hidden layer and `tanh` activation as

```
x_input = tf.keras.layers.Input(shape=(1,), name="x_input")
#Define the shape of the inputs

l1 = tf.keras.layers.Dense(40,activation="tanh")(x_input)
#Define a layer with 40 neurons and tanh activation.

u_output = tf.keras.layers.Dense(1)(l1)
#Define a layer with 1 neuron to be the output (no activation)

u_model = tf.keras.Model(inputs=x_input,outputs = u_output)
#Create the model
```

- As Tensorflow isn't really made for 1D problems, it understands the input to be $\mathtt{N}\times 1$ in shape, and this yields an $\mathtt{N}\times 1$ output.

```
x = tf.constant([[1.],[2.],[3.]])  #Define a 3x1 tensor

u_model(x)   #Evaluate the model
#<tf.Tensor: shape=(3, 1), dtype=float32, numpy=
#array([[-0.25351194],
#       [-0.45885777],
#       [-0.596128  ]], dtype=float32)>
```

- The command `u_model.summary()` gives information about the network (architecture, number of variables, etc.).

```
In [86]: u_model.summary()
Model: "functional_53"
```

Layer (type)	Output Shape	Param #
x_input (InputLayer)	(None, 1)	0
dense_26 (Dense)	(None, 40)	80
dense_27 (Dense)	(None, 1)	41

```
Total params: 121 (484.00 B)
Trainable params: 121 (484.00 B)
Non-trainable params: 0 (0.00 B)
```

- We can also access information about the layers themselves using `layers[i]` for the `i`-th layer. For example,

```
u_model.layers[1]
# <Dense name=dense_26, built=True>

u_model.layers[1].weights
#[<KerasVariable shape=(1, 40), dtype=float32, path=dense_26/
    kernel>,
# <KerasVariable shape=(40,), dtype=float32, path=dense_26/bias>]
```

- Tensorflow has plenty of in-built layers, but sometimes we need to define one ourselves.
- Let us define a new layer that applies a cutoff function to our NN.

```
class bc_layer(tf.keras.layers.Layer): ###Name the BC layer
    def __init__(self,a,b): ##Initialisation - include self, any
        parameters
        super(bc_layer,self).__init__() ##Define objects contained in
            the class
        self.a = a
        self.b = b
        #a and b will be the cutoff points, so u(a)=u(b)=0

    def call(self,inputs):  ##Define how constructs the output
        x,u1 = inputs # The inputs are a pair [x,u1]
        cut = (x-self.a)*(x-self.b) #We define the cutoff function
        output = cut*u1 #Apply the cutoff function
        return output
```

- Once we have defined our layer, we can use it just like any other layer.
- We can define our NN using our cutoff function as follows

```
x_input = tf.keras.layers.Input(shape=(1,), name="x_input")
#Define the shape of the inputs

l1 = tf.keras.layers.Dense(40,activation="tanh")(x_input)
#Define a layer with 40 neurons and tanh activation.

u_no_cutoff = tf.keras.layers.Dense(1)(l1)
#Define a layer with 1 neuron to be the output (no activation)

u_output = bc_layer(0.,1.)([x_input,u_no_cutoff])
#Apply cutoff. I specify 0. and 1. as the cutoff points,
#the input is the x-value and the result of the previous layer.

u_model = tf.keras.Model(inputs=x_input,outputs = u_output)
#Create the model
```

- When doing more "black-box" problems (e.g. interpolation), training the network is simple. You must choose the optimiser, loss function, give data, and so on. The training loop is defined like this.

```
optimizer = tf.keras.optimizers.Adam(learning_rate=10**-3)
#Define the optimiser, there are typically many parameters to choose
    from
u_model.compile(optimizer=optimizer,loss="mse")
#The model has to be compiled, specifying the loss, before training
history = u_model.fit(
    x = x_data,  #My x_data already exists
    y = y_data,  #As does my y)data
    epochs=1000, #How many epochs
    batch_size=32, #Batch size. No. Batches=epochs/batch_size
    validation_data =[x_val,y_val]
    )
#This defines the training loop
```

- The `history` object contains within it the metrics that you have used. By default, it will save the loss and validation loss (if present). Any other metrics specified will be contained.
- The object `history.history["loss"]` will return a list of the loss at every epoch, whilst `history.history["val_loss"]` returns the validation loss. This is useful to visualise your results afterwards.
- The object is returned as a list of floats.

Defining a loss function

- Tensorflow is not designed to solve problems with PDEs, so defining a loss function involving derivatives of the neural network is not a "native" function.
- There are many ways to do so, I will focus on just one method.
- First, we have to understand a little more how the "default" settings work.
- We compiled the model in our interpolation problem with

```
u_model.compile(optimizer=optimizer,loss="mse")
```

- The `mse` loss has the following structure:

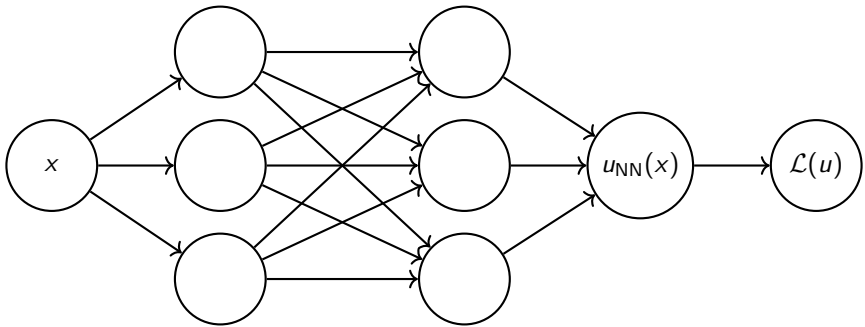
```
def mse(y_true,y_pred):
    return tf.reduce_mean((y_true-y_pred)**2)
```

- It compares the output of the neural network (`y_pred`) with the corresponding data (`y_true`).

- Our aim will be to construct a neural network, where the last layer is the loss itself. Then, our `y_pred` will be $\mathcal{L}(u)$, and the optimiser will aim to minimise the loss if we employ the following:

```
def my_loss(y_true,y_pred):
    return y_pred

u_model.compile(optimizer=optimizer,loss=my_loss)
```



- I do this by defining two models: I have `u_model`, which is my solution, defined however I wish.
- Then, I define a `loss_model`, which contains `u_model`, and outputs the loss.
- I need to use a custom layer to define the loss, which I call `loss_layer`
- Let's consider an example for solving the ODE, $u'(x) = \frac{x}{2}$ with $u(0) = 0$ on $(0, 1)$.
- I'll use a PINNs style loss with N MC points and a penality of weight w on the boundary,

$$\mathcal{L}(u) = wu(0)^2 + \frac{1}{N}\sum_{n=1}^{N}\left(u'(x_n) - 2x_n\right)^2.$$

```
class loss_layer(tf.keras.layers.Layer): ###Name the layer
    def __init__(self,u_model,n,w): #Initialisation with parameters
        super(loss_layer,self).__init__() ##Define class' objects
        self.n=n #This will be the number of points for MC
        self.w = w #The weight for the boundary term
        self.u_model=u_model  #Load u_model into loss_layer
        self.x0 = tf.constant([[0.]]) # Boundary x
    def call(self,inputs):  #The input plays no role
        x = tf.random.uniform([self.n,1]) #Take MC Sample
        with tf.GradientTape() as t1:
            t1.watch(x)
            u=self.u_model(x)  #Evaluate u
        du = t1.gradient(u,x)  #Evaluate derivative
        loss = self.w*self.u_model(self.x0)**2+tf.reduce_mean((du-2*x)
            **2) #Evaluate
        return loss
```

- Once I have my loss layer defined, I can define my loss function just like any other model in Tensorflow.

```
fake_input = tf.keras.layers.Input(shape=(1,), name="fake_input")
#Define the shape of the inputs for the loss model

loss_output = loss_layer(u_model,100,1)(fake_input)
#Apply the loss layer

loss_model = tf.keras.Model(inputs=fake_input,outputs = loss_output)
#Create the model
```

- The "input" to the network has no role, but we have to specify one for the training loop as `.fit` needs data.

- We then train the loss model just like any other model.

```
optimizer = tf.keras.optimizers.Adam(learning_rate=10**-3) #Define
    optimizer

def my_loss(y_true,y_pred): #Define the loss function used in .fit
    return y_pred

loss_model.compile(optimizer=optimizer,loss=my_loss) #compile

history = loss_model.fit(x = tf.constant([1.]), #Fake input
                         y = tf.constant([1.]), #Fake output
                         epochs = 5000)         #No/ epochs
```

- If we don't have "data", how do we measure a validation loss?
- We can apply the same idea. I will change my loss layer to evaluate the loss over a validation set as well.
- When using MC, remember, it is better to use a fixed validation - we avoid noise in the validation set.
- We introduce a object, `self.xval` in `__init__`,

```
self.xval = tf.random.uniform([int(n/5),1])
```

```
def call(self,inputs):  #The input plays no role
    x = tf.random.uniform([self.n,1]) #Take MC Sample
    with tf.GradientTape() as t1:
        t1.watch(x)
        u=self.u_model(x)  #Evaluate u
    du = t1.gradient(u,x)  #Evaluate derivative
    bc_loss = self.w*self.u_model(self.x0)**2
    loss = bc_loss +tf.reduce_mean((du-2*x)**2) #Evaluate ODE loss
    with tf.GradientTape() as t1:
        t1.watch(self.xval)
        u_val=self.u_model(self.xval)  #Evaluate validation
    du_val = t1.gradient(u_val,self.xval)  #Evaluate validation
        derivative
    loss_val =  bc_loss +tf.reduce_mean((du_val-2*self.xval)**2) #
        Evaluate val loss
    return tf.concat([loss,loss_val],axis=-1)
```

- Now the output of `loss_model` is a shape $[2, 1]$ tensor. My loss corresponds to `y_pred[0,0]` whilst the validation is `y_pred[0,1]`.
- I need to redefine the loss that goes into fit, and I define my validation as a metric.

```
def my_loss(y_true,y_pred):
    return y_pred[0,0]

def my_val(y_true,y_pred):
    return y_pred[0,1]

loss_model.compile(optimizer=optimizer,loss=my_loss,metrics=[my_val])
```

- I then train just as before. If I call `history.history["my_val"]`, I can access the values of the validation loss per epoch.

- It is also possible to define the metrics as distinct layers, and concatenate the outputs.
- The same idea holds if you want to measure distinct components of the loss. Let us consider a 2D PINN problem, and see how we can do this.
- We will consider $\Delta u(x, y) = f(x, y)$ with Dirichlet boundary conditions, where the exact solution is chosen so that f and u can be expressed as simple trigonometric functions.
- We take our domain to be $\Omega = \{(x, y) \in \mathbb{R}^2 : x^2 + y^2 < 1\}$.
- We can sample the boundary by taking points $(\cos(t), \sin(t))$, with t uniformly distributed in $[0, 2\pi]$.
- We can sample the interior uniformly by taking points $\sqrt{s}(\cos(t), \sin(t))$ with t uniform on $[0, 2\pi]$ and s uniform on $[0, 1]$.

- I've defined layers for the PDE part of the loss, the BC part, and another to measure the H^1-error. I have another layer that stacks the three outputs into a single [3] tensor.

```
fake_input = tf.keras.layers.Input(shape=(1,), name="fake_input")
#Define the shape of the inputs for the loss model
loss_PDE = loss_PDE_layer(u_model,n1)(fake_input)
#Define the PDE loss
loss_bc = loss_bc_layer(u_model,n2)(fake_input)
#Define the BC loss
metric_h1 = metric_h1_layer(u_model)(fake_input)
#Define the metric
loss_output = my_stack_layer()([loss_PDE,loss_bc,metric_h1])
#Final output
loss_model = tf.keras.Model(inputs=fake_input,outputs = loss_output)
#Create the model
```

- I now define my losses and metrics used in `.fit`
- Recall, the output of `loss_model` is `[loss_PDE,loss_bc,metric_h1]`

```
def my_loss(y_true,y_pred):  #The full loss, with the weight w
    return y_pred[0]+w*y_pred[1]

def pde_loss(y_true,y_pred): #Only measure the PDE component
    return y_pred[0]

def bc_loss(y_true,y_pred): #Only measure the bc component
    return y_pred[1]

def my_val(y_true,y_pred): #measure the H1-error
    return y_pred[2]

loss_model.compile(optimizer=optimizer,loss=my_loss,metrics=[pde_loss,
    bc_loss,my_val])
#Compile with loss and metrics I want to measure.
```

Adding trainable variables

- In a custom layer, you may want to add trainable variables as well.
- If added correctly, in `.fit`, the optimiser will automatically optimise them as well.
- We will consider a simple inverse problem as before - We have noisy data (x_j, u_j), and we know that u satisfies the ODE $ku''(x) = f(x)$ and $u(0) = u(1) = 0$. We know f, but we don't know k.
- We want to introduce k as a trainable variable in our loss function.
- We will modify our loss layer to include a new trainable weight.

- We modify our 1D PINNs loss layer via

```
class loss_layer_PDE(tf.keras.layers.Layer): ###Name the layer
    def __init__(self,u_model,n,w): #Initialisation with parameters,
        u_model
            [......]
    def build(self,inputs):
        self.k = self.add_weight(shape=[1],
                                 initializer=tf.keras.initializers.
                                     RandomUniform(minval=0,maxval=0))
    def call(self,inputs):  #The input plays no role
        [......]
        loss_PDE = tf.reduce_mean((self.k*ddu-rhs(x))**2) #Evaluate
            ODE loss
        loss_bc = self.w*tf.reduce_sum(self.u_model(self.x0)**2)
        return loss_bc+loss_PDE
```

- To call the value of `k`, we can call it like any object in a class. If the layer index of `loss_layer_PDE` is `i`, then

```
loss_model.layers[i].k[0]
```

yields the value of k.

- If you aren't sure which layer index corresponds to which layer, use

```
loss_model.layers
#[<InputLayer name=fake_input, built=True>,
# <loss_layer_PDE name=loss_layer_pde_2, built=True>,
# <loss_layer_interpolation name=loss_layer_interpolation_2, built=
    True>]
```

- Note: Different versions of tensorflow can change the order, even with the same code.

- Sometimes you will want to do operations beyond parameter updates and metrics during a training loop.
- Some examples are stopping training if the validation starts increasing, reducing the learning rate if there is no improvement, saving the best weights during training etc.
- These are generally implemented via *Callbacks*.
- Tensorflow has plenty built in, and we can also define custom callbacks.

- As a first example, let's look at how to use an "off-the-shelf" callback.
- We will use the `ReduceLROnPlateau` callback. It measures a quantity given by you (usually the validation loss), and if it stops decreasing, it decreases the learning rate.

```
callbacks = [tf.keras.callbacks.ReduceLROnPlateau(monitor="my_val",
                                                  factor=0.8,
                                                  patience=50,
                                                  cooldown=50)]


history = loss_model.fit(x = tf.constant([[1.]]),
                         y = tf.constant([1.]),
                         epochs = 5000,
                         callbacks = callbacks)
```

- Let's define a custom callback. At logarithmically spaced intervals, we will evaluate the error by comparing to the exact solution.
- This may be advisable, because it can be more expensive than evaluating the loss itself and it doesn't need to be done at every iteration.
- We will define a `rate`, and evaluate the H^1-error at iterations roughly of size $\texttt{rate}^k$.
- The callback will save two lists inside it, `error_list`, recording the errors, and `its_list`, recording at which iteration it does the measurement.
- We can access these lists after training.

- We will start by considering just what happens in `init`.

```
class measure_error(tf.keras.callbacks.Callback):
    def __init__(self,u_model,rate):
        super(measure_error,self).__init__() #Up to here, like a
            custom layer or general class
        self.u_model = u_model
        self.xtest = tf.constant([[(i+0.5)/200] for i in range(200)])
        self.next_it=1        #Next iteration to record at
        self.rate=rate        #Rate of recording
        self.error_list = [] #Will be the list of recorded errors
        self.its_list=[]      #Will be the list of recorded iterations
```

- Next, we consider what is happening when the callback actually acts.

```
class measure_error(tf.keras.callbacks.Callback):
[......]
    def on_epoch_end(self,epoch,logs=None):
        if epoch>self.next_it: #Test iteration number.
            self.its_list+=[epoch] #Add the iteration
            self.next_it = int(self.rate*self.next_it)+1  #Find next
                iteration
            with tf.GradientTape() as t1:
                t1.watch(self.xtest)
                u_err = self.u_model(self.xtest)-u_exact(self.xtest)
            du_err = t1.gradient(u_err,self.xtest) #Evaluate the error
            self.error_list+= [tf.reduce_mean(du_err**2+u_err**2)]
            #Append to error list
```

- Then we just add the callback like we would an "off-the-shelf" one.

```
measure_callback = measure_error(u_model,1.1)

history = loss_model.fit(x = tf.constant([1.]),
                         y = tf.constant([1.]),
                         epochs = 5000,
                         callbacks = [measure_callback])
```

- We can call the iteration and error list easily

```
measure_callback.error_list
measure_callback.its_list
```