In [14]:
```
!pip install pyDOE
```

```
Requirement already satisfied: pyDOE in /usr/local/lib/python3.10/dist-packages (0.3.8)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from pyDOE) (1.26.4)
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-packages (from pyDOE) (1.13.1)
```

In [15]:
```
import tensorflow as tf
import numpy as np
from pyDOE import lhs
import matplotlib.pyplot as plt
```

In [16]:
```
# Initial condition
u0 = 0.5

# Boundaries of the computational domain
t0, tfinal = 0.0, 1.0
```

In [17]:
```
# Define the network
def build_model(nr_units=20, nr_layers=4, summary=True):

  inp = b = tf.keras.layers.Input(shape=(1,))

  for i in range(nr_layers):
    b = tf.keras.layers.Dense(nr_units, activation='tanh')(b)
  out = tf.keras.layers.Dense(1, activation='linear')(b)

  model = tf.keras.models.Model(inp, out)

  if summary:
    model.summary()

  return model
```

In [18]:
```
model = build_model()
```

Model: "functional_1"

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 1)	0
dense_5 (Dense)	(None, 20)	40
dense_6 (Dense)	(None, 20)	420
dense_7 (Dense)	(None, 20)	420
dense_8 (Dense)	(None, 20)	420
dense_9 (Dense)	(None, 1)	21

Total params: 1,321 (5.16 KB)

Trainable params: 1,321 (5.16 KB)

Non-trainable params: 0 (0.00 B)

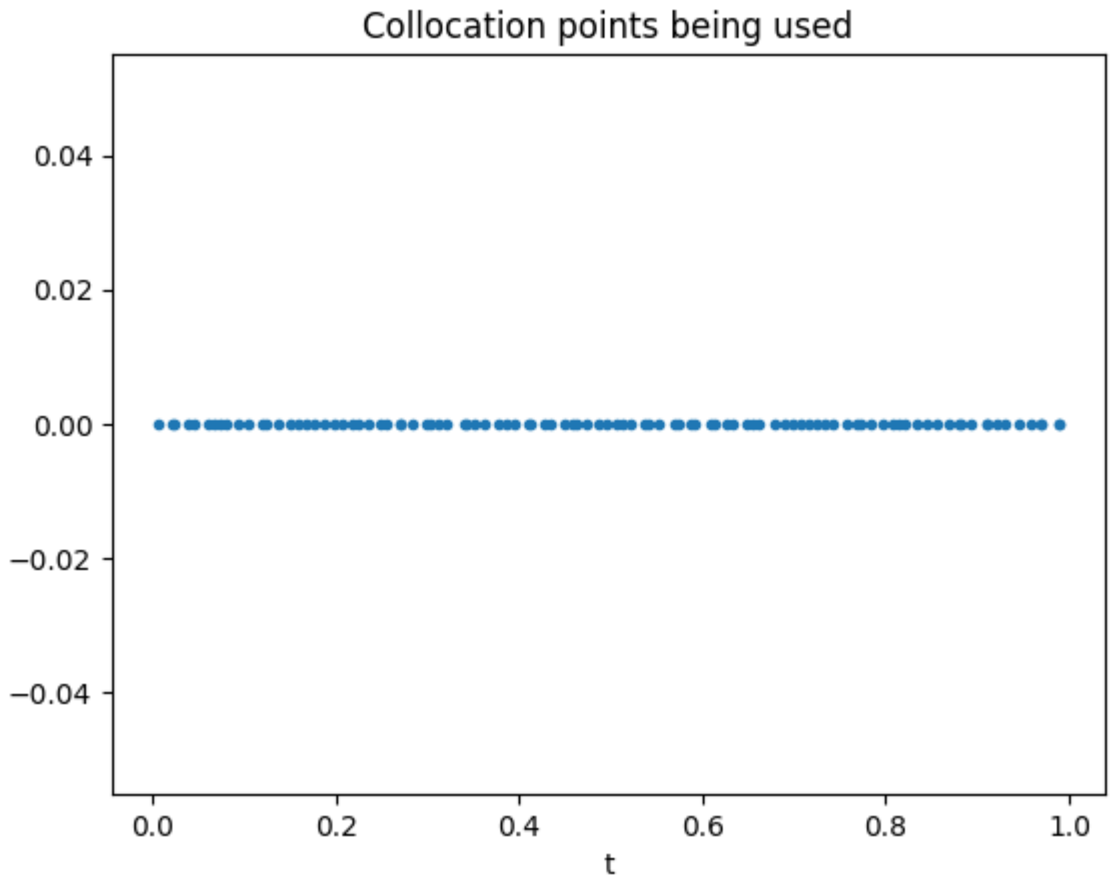
In [19]:
```
# Define the collocation points over the domain [t_0, t_1]
def defineCollocationPoints(t_bdry, N_de=100):

  # Sample points where to evaluate the ODE
  ode_points = t_bdry[0] + (t_bdry[1] - t_bdry[0])*lhs(1, N_de)

  return ode_points
```

In [20]:
```
# Define the collocation points
de_points = defineCollocationPoints([t0, tfinal], 100)
```

In [21]:
```
plt.plot(de_points[:,0], 0*de_points[:,0],'.')
plt.xlabel('t')
plt.title('Collocation points being used')
plt.savefig('CollocationPoints1D.png')
```



In [22]:
```
# The main training function
@tf.function
def train_network(t, model, gamma=1):

    # Outer gradient for tuning network parameters
    with tf.GradientTape() as tape:

      # Inner gradient for derivatives of u wrt x and t
      with tf.GradientTape() as tape2:
        tape2.watch(t)
        u = model(t)

      # Derivative of the neural network solution
      ut = tape2.gradient(u, t)

      # Define the differential equation loss
      eqn = ut + u
      DEloss = tf.reduce_mean(eqn**2)

      # Define the initial value loss
      u0_pred = model(np.array([[t0]]))
      IVloss = tf.reduce_mean((u0_pred - u0)**2)

      # Composite loss function
      loss = DEloss + gamma*IVloss

    grads = tape.gradient(loss, model.trainable_variables)
    return loss, grads
```

In [23]:
```
def PINNtrain(de_points, model, epochs=1000):

  # Total number of collocation points
  N_de = len(de_points)

  # Batch size
  bs_de = N_de

  # Learning rate
  lr_model = 1e-3

  epoch_loss = np.zeros(epochs)
  nr_batches = 0

  # Generate the tf.Dataset for the differential equations points
  ds = tf.data.Dataset.from_tensor_slices(de_points.astype(np.float32))
  ds = ds.cache().shuffle(N_de).batch(bs_de)

  # Generate the model
  opt = tf.keras.optimizers.Adam(lr_model)

  # Main training loop
  for i in range(epochs):

    # Training for that epoch
    for des in ds:

      # Train the network and gradient descent step
      loss, grads = train_network(des, model)
      opt.apply_gradients(zip(grads, model.trainable_variables))

      epoch_loss[i] += loss
      nr_batches += 1

    # Get total epoch loss
    epoch_loss[i] /= nr_batches
    nr_batches = 0

    if (np.mod(i, 100)==0):
      print(f"Loss {i}th epoch: {epoch_loss[i]: 6.4f}")

  return epoch_loss
```

In [24]:
```
epochs = 5000
loss = PINNtrain(de_points, model, epochs)
```

```
Loss 0th epoch:  0.2577
Loss 100th epoch:  0.0009
Loss 200th epoch:  0.0005
Loss 300th epoch:  0.0002
Loss 400th epoch:  0.0001
Loss 500th epoch:  0.0000
Loss 600th epoch:  0.0000
Loss 700th epoch:  0.0000
Loss 800th epoch:  0.0000
Loss 900th epoch:  0.0000
Loss 1000th epoch:  0.0000
Loss 1100th epoch:  0.0000
Loss 1200th epoch:  0.0000
Loss 1300th epoch:  0.0000
Loss 1400th epoch:  0.0000
Loss 1500th epoch:  0.0000
Loss 1600th epoch:  0.0000
Loss 1700th epoch:  0.0000
Loss 1800th epoch:  0.0000
Loss 1900th epoch:  0.0000
Loss 2000th epoch:  0.0000
Loss 2100th epoch:  0.0000
Loss 2200th epoch:  0.0000
Loss 2300th epoch:  0.0001
Loss 2400th epoch:  0.0000
Loss 2500th epoch:  0.0000
Loss 2600th epoch:  0.0000
Loss 2700th epoch:  0.0000
Loss 2800th epoch:  0.0000
Loss 2900th epoch:  0.0000
Loss 3000th epoch:  0.0000
Loss 3100th epoch:  0.0000
Loss 3200th epoch:  0.0000
Loss 3300th epoch:  0.0000
Loss 3400th epoch:  0.0000
Loss 3500th epoch:  0.0000
Loss 3600th epoch:  0.0001
Loss 3700th epoch:  0.0000
Loss 3800th epoch:  0.0000
Loss 3900th epoch:  0.0000
Loss 4000th epoch:  0.0000
Loss 4100th epoch:  0.0000
Loss 4200th epoch:  0.0000
Loss 4300th epoch:  0.0000
Loss 4400th epoch:  0.0000
Loss 4500th epoch:  0.0000
Loss 4600th epoch:  0.0000
Loss 4700th epoch:  0.0000
Loss 4800th epoch:  0.0000
Loss 4900th epoch:  0.0000
```

In [25]:
```
# Grid where to evaluate the model
m = 100
t = np.linspace(t0, tfinal, m)

# Model prediction
u = model(np.expand_dims(t, axis=1))[:,0]
```

In [26]:
```
# Plot the solution

# This is the exact solution
uexact = u0*np.exp(-t)

fig = plt.figure(figsize=(21,7))

# Plot the numerical and exact solutions
plt.subplot(131)
plt.plot(t, u)
plt.plot(t, uexact,'k--')
plt.grid()
plt.xlabel('t')
plt.ylabel('u')
plt.legend(['NN solution','Exact solution'])

# Plot the error
plt.subplot(132)
plt.plot(t, u-uexact)
plt.grid()
plt.xlabel('t')
plt.ylabel('error')

# Plot the loss function
plt.subplot(133)
plt.semilogy(np.linspace(1, epochs, epochs),loss)
plt.grid()
plt.xlabel('epochs')
plt.ylabel('loss')

plt.savefig('MATH3030DecayProblem.png')
```