Knapp, Stephen

Deep Learning Assignment 8 - Pytorch

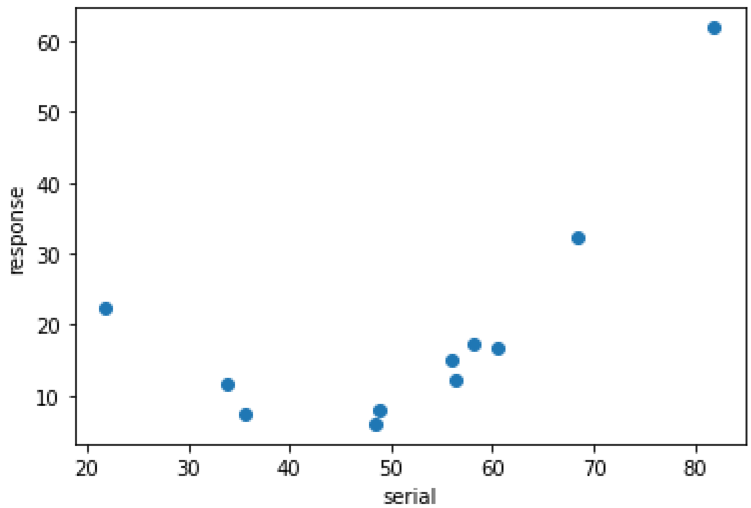
Set-up and Imports

In [0]:
```
import torch
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
from IPython.display import Image
from torch.autograd import Variable
import torch.nn as nn
import h5py
import torch.optim as optim
```

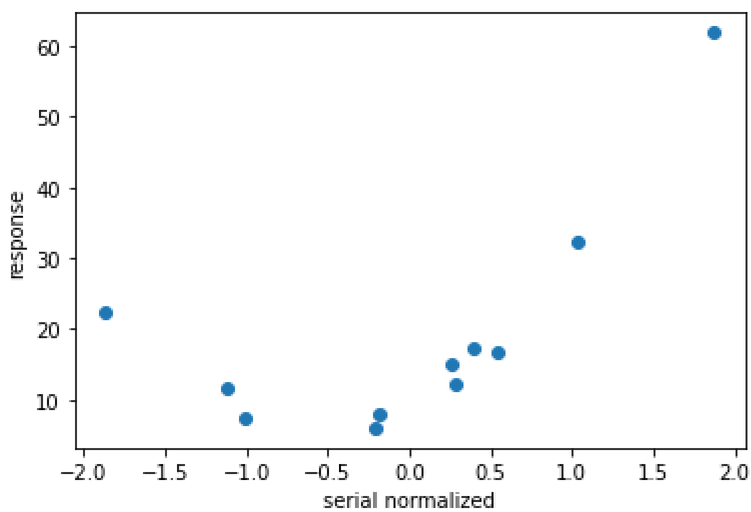
Problem 1: Manual model

In [0]:
```
torch.set_printoptions(edgeitems=2)
t_c = [7.3, 15.1, 17.2, 61.9, 12.3, 8.0, 11.6, 22.5, 6.0, 16.6, 32.2]
t_f = [35.7, 55.9, 58.2, 81.9, 56.3, 48.9, 33.9, 21.8, 48.4, 60.4, 68.4]
t_fn = (t_fo - np.mean(t_fo)) / np.std(t_fo)
t_c = torch.tensor(t_c)
t_f = torch.tensor(t_f)
t_fn = torch.tensor(t_fn)
```

In [304]:
```
plt.ylabel("response")
plt.xlabel("serial")
plt.plot(t_f, t_c,"o")
plt.show()
```

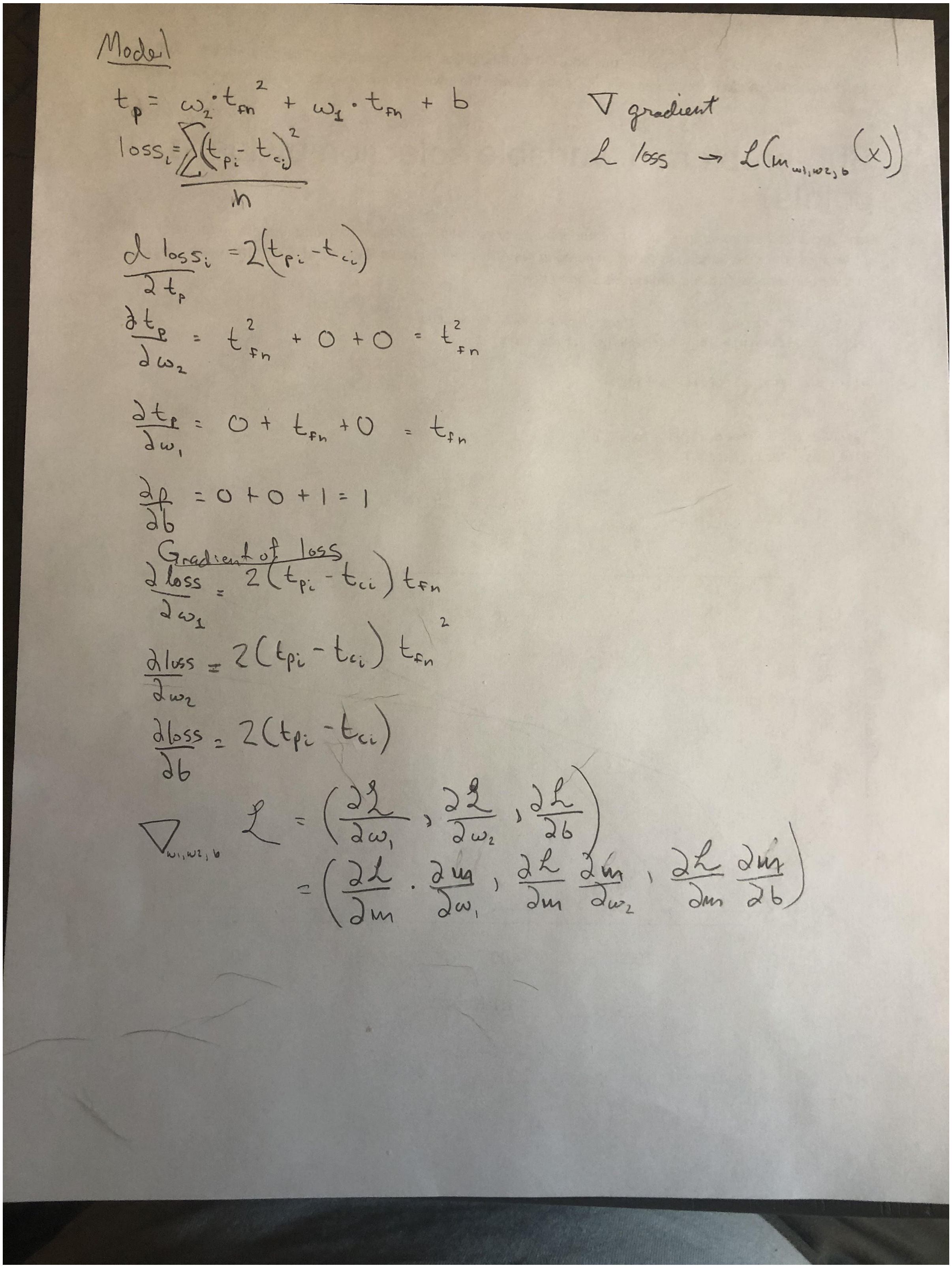


In [305]:
```
plt.ylabel("response")
plt.xlabel("serial normalized")
plt.plot(t_fn, t_c,"o")
plt.show()
```



In [306]:
```
Image("handcalcs.jpg")
```

Out[306]:



Model

$$t_p = \omega_2 \cdot t_{fn}^2 + \omega_1 \cdot t_{fn} + b$$

$$loss_i = \frac{\sum (t_{p_i} - t_{c_i})^2}{n}$$

∇ gradient

$\mathcal{L}$ loss $\rightarrow \mathcal{L}(m_{w1,w2,b}(x))$

$$\frac{d\, loss_i}{\partial t_p} = 2(t_{p_i} - t_{c_i})$$

$$\frac{\partial t_p}{\partial \omega_2} = t_{fn}^2 + 0 + 0 = t_{fn}^2$$

$$\frac{\partial t_p}{\partial \omega_1} = 0 + t_{fn} + 0 = t_{fn}$$

$$\frac{\partial p}{\partial b} = 0 + 0 + 1 = 1$$

Gradient of loss

$$\frac{\partial\, loss}{\partial \omega_1} = 2(t_{p_i} - t_{c_i})\, t_{fn}$$

$$\frac{\partial\, loss}{\partial \omega_2} = 2(t_{p_i} - t_{c_i})\, t_{fn}^2$$

$$\frac{\partial\, loss}{\partial b} = 2(t_{p_i} - t_{c_i})$$

$$\nabla_{w1,w2,b} \mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial \omega_1}, \frac{\partial \mathcal{L}}{\partial \omega_2}, \frac{\partial \mathcal{L}}{\partial b}\right)$$

$$= \left(\frac{\partial \mathcal{L}}{\partial m} \cdot \frac{\partial m}{\partial \omega_1}, \frac{\partial \mathcal{L}}{\partial m} \frac{\partial m}{\partial \omega_2}, \frac{\partial \mathcal{L}}{\partial m} \frac{\partial m}{\partial b}\right)$$

In [0]:
```
# The model is quadratic
def model(t_f, w1, w2, b):
    return w2 * t_f**2 + w1 * t_f + b

#Loss function is mean square error. t_p stands for t_predicted
def loss_fn(t_p, t_c):
    squared_diffs = (t_p - t_c)**2
    return squared_diffs.mean()
```

My model is quadratic and therefore will have three trainable parameters: the coefficients for t_f^2 (w2) and t_f (w1) as well as a constant (b).

In [308]:
```
w1 = torch.ones(1)
w2 = torch.ones(1)
b = torch.zeros(1)
t_p = model(t_fn, w1, w2, b)
print(t_p)
```

```
tensor([ 3.2118e-03,  3.2074e-01,  5.5782e-01,  5.3932e+00,  3.5902e-01,
        -1.4805e-01,  1.2867e-01,  1.6250e+00, -1.6697e-01,  8.2303e-01,
         2.1043e+00], dtype=torch.float64)
```

In [309]:
```
# The loss function for those particular values of `(w,b)` would be:
loss = loss_fn(t_p, t_c)
print(loss)
```

```
tensor(519.1525, dtype=torch.float64)
```

In [0]:
```
def dloss_fn(t_p, t_c):
        dsq_diffs = 2 * (t_p - t_c)
        return dsq_diffs
```

In [0]:
```
# Since t_p = model(w1, w2, b), the derivatives of t_p with respect to w1, w2
 and b are given by:
def dmodel_db(t_f, w1, w2, b):
        return 1.0

def dmodel_dw1(t_f, w1, w2, b):
    return t_f

def dmodel_dw2(t_f, w1, w2, b):
    return t_f**2
```

In [0]:
```
# The function returning the gradient of the loss with respect to w1, w2 and b
is:
def grad_fn(t_f, t_c, t_p, w1, w2, b):
  dloss_dw1 = dloss_fn(t_p, t_c) * dmodel_dw1(t_f, w1, w2, b)
  dloss_dw2 = dloss_fn(t_p, t_c) * dmodel_dw2(t_f, w1, w2, b)
  dloss_db = dloss_fn(t_p, t_c) * dmodel_db(t_f, w1, w2, b)
  return torch.stack([dloss_dw1.mean(), dloss_dw2.mean(),dloss_db.mean()])
```

In [0]:

```
def training_loop(n_epochs, learning_rate, params, t_f, t_c, print_params=True):
    for epoch in range(1, n_epochs + 1):
        w1, w2, b = params

        t_p = model(t_f, w1, w2, b)  #  Forward pass
        loss = loss_fn(t_p, t_c)
        gradient = grad_fn(t_f, t_c, t_p, w1, w2, b)  #  Backward pass

        params = params - learning_rate * gradient

        if epoch in {1, 2, 3, 10, 11, 99, 100, 1000, 2000, 5000, 10000, 20000}:
            print('Epoch %d, Loss %f' % (epoch, float(loss))) #  Periodic logging
            if print_params:
                print('    Params:', params)
                print('    Grad:  ', gradient)
        if epoch in {4, 12, 101}:
            print('...')

        if not torch.isfinite(loss).all():
            break  # <3>

    return params
```

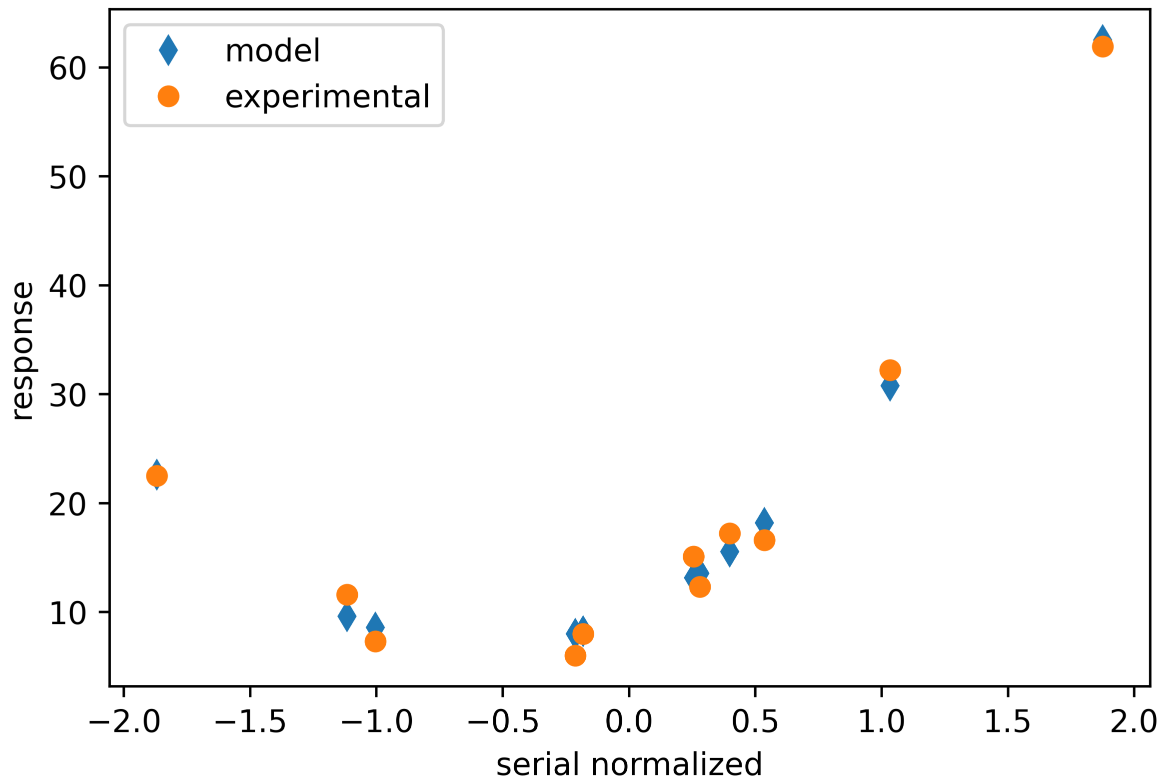
In [314]:
```
# provide initial values for `w1` w2 and `b` and only then train the model:
params = training_loop(
    n_epochs = 5000,
    learning_rate = 1e-2,
    params = torch.tensor([1.0, 1.0, 0.0]),
    t_f = t_fn,
    t_c = t_c,
    print_params = True)
```

```
Epoch 1, Loss 519.152499
    Params: tensor([1.1771, 1.6094, 0.3631], dtype=torch.float64)
    Grad:   tensor([-17.7110, -60.9372, -36.3091], dtype=torch.float64)
Epoch 2, Loss 467.244234
    Params: tensor([1.3518, 2.1803, 0.7067], dtype=torch.float64)
    Grad:   tensor([-17.4658, -57.0949, -34.3642], dtype=torch.float64)
Epoch 3, Loss 421.151504
    Params: tensor([1.5240, 2.7152, 1.0321], dtype=torch.float64)
    Grad:   tensor([-17.2187, -53.4896, -32.5350], dtype=torch.float64)
...
Epoch 10, Loss 213.361095
    Params: tensor([2.6593, 5.6206, 2.8808], dtype=torch.float64)
    Grad:   tensor([-15.4683, -33.7696, -22.4303], dtype=torch.float64)
Epoch 11, Loss 195.044318
    Params: tensor([2.8115, 5.9366, 3.0939], dtype=torch.float64)
    Grad:   tensor([-15.2194, -31.6042, -21.3063], dtype=torch.float64)
...
Epoch 99, Loss 5.465423
    Params: tensor([9.3215, 9.9964, 8.2734], dtype=torch.float64)
    Grad:   tensor([-2.7339,  0.5921, -1.7935], dtype=torch.float64)
Epoch 100, Loss 5.355987
    Params: tensor([9.3483, 9.9905, 8.2911], dtype=torch.float64)
    Grad:   tensor([-2.6781,  0.5925, -1.7695], dtype=torch.float64)
...
Epoch 1000, Loss 2.065094
    Params: tensor([10.6003,  9.3226,  9.8318], dtype=torch.float64)
    Grad:   tensor([ 1.2133e-05,  6.6221e-05, -1.3598e-04], dtype=torch.float
64)
Epoch 2000, Loss 2.065094
    Params: tensor([10.6003,  9.3226,  9.8320], dtype=torch.float64)
    Grad:   tensor([ 4.0399e-10,  2.1985e-09, -4.5144e-09], dtype=torch.float
64)
Epoch 5000, Loss 2.065094
    Params: tensor([10.6003,  9.3226,  9.8320], dtype=torch.float64)
    Grad:   tensor([ 8.5184e-14,  8.6678e-14, -8.8172e-14], dtype=torch.float
64)
```

In [315]:
```
print(params)
```

```
tensor([10.6003,  9.3226,  9.8320], dtype=torch.float64)
```

In [316]:
```
%matplotlib inline
from matplotlib import pyplot as plt

t_p = model(t_fn, *params)
# Besides the original training data, we will plot prediction `t_p` for every
 input `t_f`
fig = plt.figure(dpi=600)
plt.xlabel("serial normalized")
plt.ylabel("response")
plt.plot(t_fn.numpy(), t_p.numpy(), 'd')
plt.plot(t_fn.numpy(), t_c.numpy(), 'o')
plt.legend(["model",'experimental'])
plt.savefig("q1plot.png", format="png")  # bookskip
```



Problem 2: Autograd with manual decent gradient

In [0]:
```
params = torch.tensor([1.0, 1.0, 0.0], requires_grad=True)   # Usually =True
```

In [318]:
```
params.grad is None
```

Out[318]:
```
True
```

In [319]:
```
loss = loss_fn(model(t_f, *params), t_c)
loss.backward()

params.grad
```

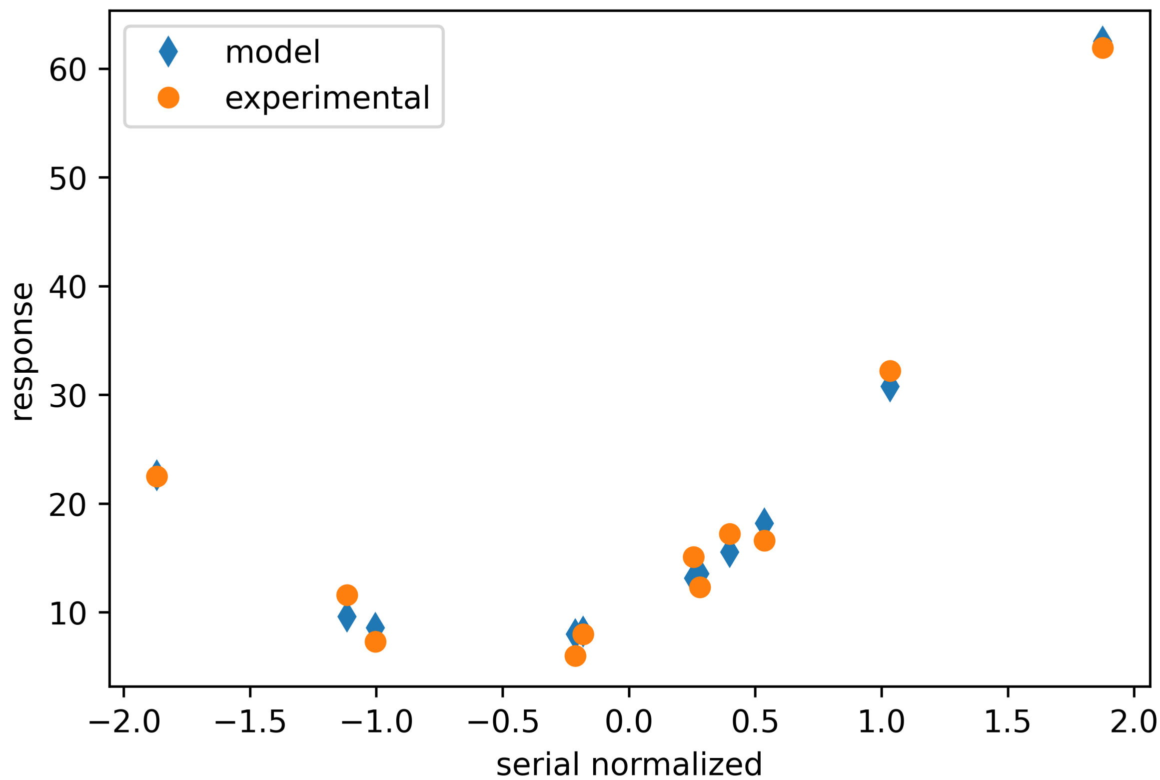
Out[319]:
```
tensor([3.6088e+05, 2.3087e+07, 5.9469e+03])
```

In [0]:
```
def training_loop(n_epochs, learning_rate, params, t_f, t_c):
    for epoch in range(1, n_epochs + 1):
        if params.grad is not None:  # This could be done at any point in the
 loop
            params.grad.zero_()      # prior to calling `loss.backward()`

        t_p = model(t_f, *params)
        loss = loss_fn(t_p, t_c)
        loss.backward()

        params = (params - learning_rate * params.grad).detach().requires_grad
_()

        if epoch % 500 == 0:
            print('Epoch %d, Loss %f' % (epoch, float(loss)))

    return params
```

In [321]:
```
params = training_loop(
    n_epochs = 5000,
    learning_rate = 1e-2,
    params = torch.tensor([1.0, 1.0, 0.0], requires_grad=True),
    t_f = t_fn,
    t_c = t_c)
```

```
Epoch 500, Loss 2.065433
Epoch 1000, Loss 2.065094
Epoch 1500, Loss 2.065094
Epoch 2000, Loss 2.065094
Epoch 2500, Loss 2.065094
Epoch 3000, Loss 2.065094
Epoch 3500, Loss 2.065094
Epoch 4000, Loss 2.065094
Epoch 4500, Loss 2.065094
Epoch 5000, Loss 2.065094
```

In [322]:
```
t_p = model(t_fn, *params)
# Besides the original training data, we will plot prediction `t_p` for every
 input `t_f`
fig = plt.figure(dpi=600)
plt.xlabel("serial normalized")
plt.ylabel("response")
plt.plot(t_fn.numpy(), t_p.detach().numpy(), 'd') # cannot call numpy() direct
ly, need detach()
plt.plot(t_fn.numpy(), t_c.numpy(), 'o')
plt.legend(["model","experimental"])
plt.savefig("q2plot.png", format="png")  # bookskip
```



Problem 3: Using optimizers

Using Optimizer SGD

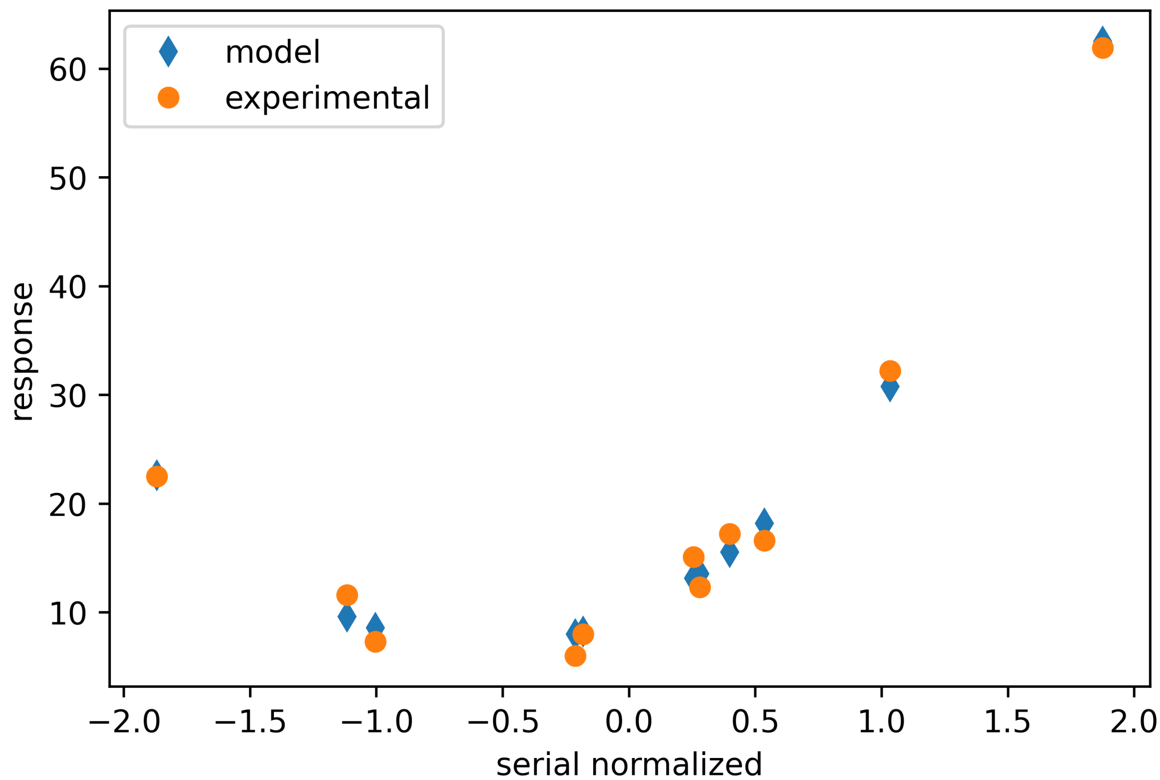
In [0]:
```
def training_loop(n_epochs, optimizer, params, t_f, t_c):
    for epoch in range(1, n_epochs + 1):
        if params.grad is not None:
          params.grad.zero_()

        t_p = model(t_f, *params)
        loss = loss_fn(t_p, t_c)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if epoch % 500 == 0:
            print('Epoch %d, Loss %f' % (epoch, float(loss)))

    return params
```

In [324]:
```
params = torch.tensor([1.0, 1.0, 0.0], requires_grad=True)
learning_rate = 1e-2
optimizer = optim.SGD([params], lr=learning_rate) # <1>

training_loop(
    n_epochs = 5000,
    optimizer = optimizer,
    params = params, # <1>
    t_f = t_fn,
    t_c = t_c)
```

```
Epoch 500, Loss 2.065433
Epoch 1000, Loss 2.065094
Epoch 1500, Loss 2.065094
Epoch 2000, Loss 2.065094
Epoch 2500, Loss 2.065094
Epoch 3000, Loss 2.065094
Epoch 3500, Loss 2.065094
Epoch 4000, Loss 2.065094
Epoch 4500, Loss 2.065094
Epoch 5000, Loss 2.065094
```

Out[324]:
```
tensor([10.6003,  9.3226,  9.8319], requires_grad=True)
```

In [325]:
```
t_p = model(t_fn, *params)
# Besides the original training data, we will plot prediction `t_p` for every
 input `t_f`
fig = plt.figure(dpi=600)
plt.xlabel("serial normalized")
plt.ylabel("response")
plt.plot(t_fn.numpy(), t_p.detach().numpy(), 'd') # cannot call numpy() direct
ly, need detach()
plt.plot(t_fn.numpy(), t_c.numpy(), 'o')
plt.legend(["model","experimental"])
plt.savefig("q3plotSDG.png", format="png")  # bookskip
```



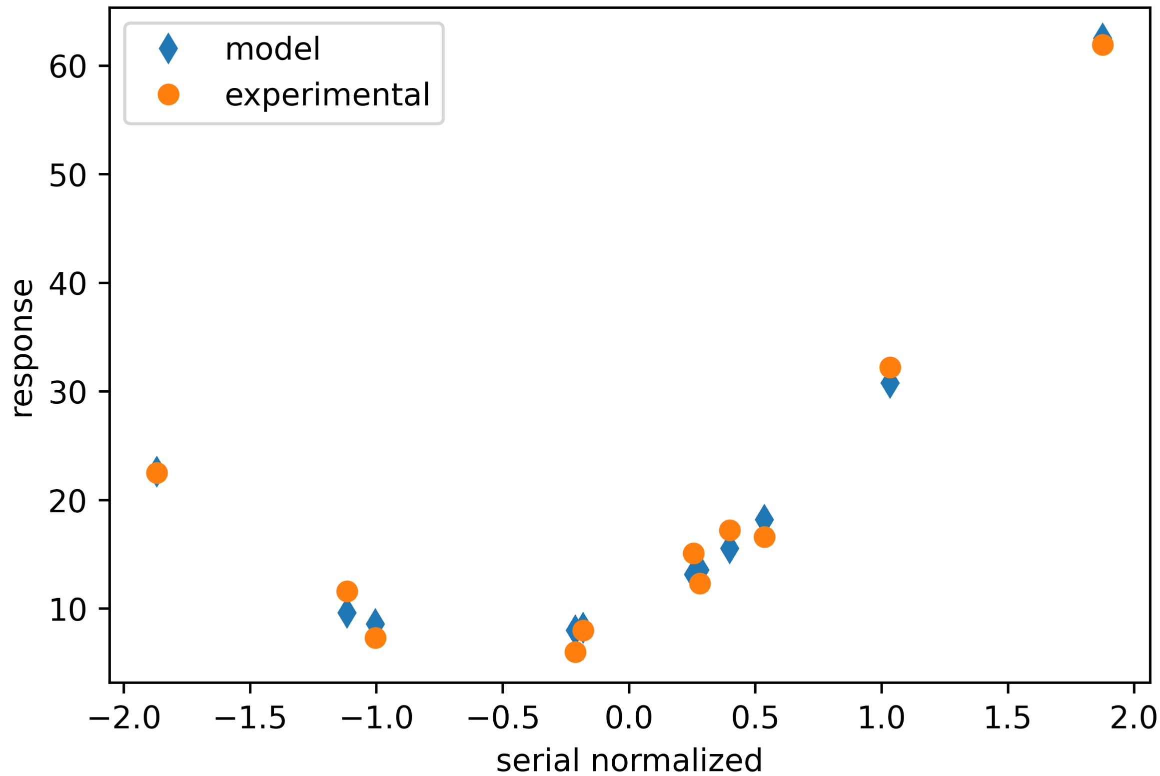
Using Optimizer Adam

In [326]:
```
params = torch.tensor([1.0, 1.0, 0.0], requires_grad=True)
learning_rate = 1e-2
optimizer = optim.Adam([params], lr=learning_rate) # <1>

training_loop(
    n_epochs = 5000,
    optimizer = optimizer,
    params = params, # <1>
    t_f = t_fn,
    t_c = t_c)
```

```
Epoch 500, Loss 142.481169
Epoch 1000, Loss 25.560932
Epoch 1500, Loss 4.264352
Epoch 2000, Loss 2.290577
Epoch 2500, Loss 2.126031
Epoch 3000, Loss 2.080929
Epoch 3500, Loss 2.067852
Epoch 4000, Loss 2.065378
Epoch 4500, Loss 2.065109
Epoch 5000, Loss 2.065095
```

Out[326]:
```
tensor([10.6003,  9.3229,  9.8313], requires_grad=True)
```

In [327]:
```
t_p = model(t_fn, *params)
# Besides the original training data, we will plot prediction `t_p` for every
 input `t_f`
fig = plt.figure(dpi=600)
plt.xlabel("serial normalized")
plt.ylabel("response")
plt.plot(t_fn.numpy(), t_p.detach().numpy(), 'd') # cannot call numpy() direct
ly, need detach()
plt.plot(t_fn.numpy(), t_c.numpy(), 'o')
plt.legend(["model","experimental"])
plt.savefig("q3plotadam.png", format="png")  # bookskip
```



Problem 4: Training and Validation Data

In [381]:
```
n_samples = t_fn.shape[0]
n_val = int(0.3 * n_samples)   # 30% of data is reserved for the validation se
t

shuffled_indices = torch.randperm(n_samples)

train_indices = shuffled_indices[:-n_val]
val_indices = shuffled_indices[-n_val:]

print(train_indices, val_indices )
```

```
tensor([ 4,  8, 10,  0,  1,  2,  6,  3]) tensor([5, 7, 9])
```

In [0]:
```
train_t_fn = t_fn[train_indices]
train_t_c = t_c[train_indices]

val_t_fn = t_fn[val_indices]
val_t_c = t_c[val_indices]
```

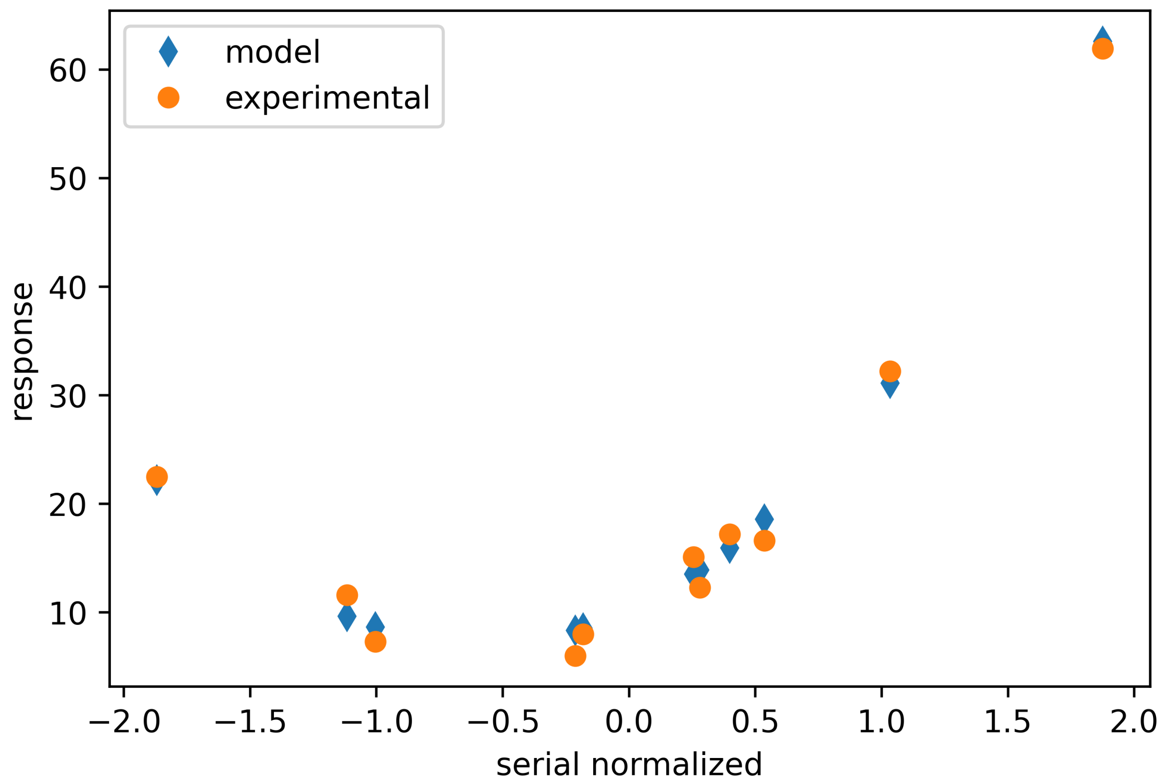
In [0]:
```
if params.grad is not None:
    params.grad.zero_()
```

In [0]:
```
def training_loop(n_epochs, optimizer, params, train_t_f, val_t_f, train_t_c,
val_t_c):
    train_loss_store = []
    val_loss_store = []

    for epoch in range(1, n_epochs + 1):

        if params.grad is not None:
          params.grad.zero_()

        train_t_p = model(train_t_f, *params) # passing model instead of indiv
idual parameters
        train_loss = loss_fn(train_t_p, train_t_c)
        train_loss_store.append(train_loss)

        val_t_p = model(val_t_f, *params) # <1>
        val_loss = loss_fn(val_t_p, val_t_c)
        val_loss_store.append(val_loss)

        optimizer.zero_grad()
        train_loss.backward() # The loss function is also passed in
        optimizer.step()

        if epoch <= 3 or epoch % 500 == 0:
          print('Epoch {}, Training loss {}, Validation loss {}'.format(
            epoch, float(train_loss), float(val_loss)))

    return [params, train_loss_store, val_loss_store]
```

In [394]:
```
params = torch.tensor([1.0, 1.0, 0.0], requires_grad=True)
learning_rate = 1e-2
optimizer = optim.SGD([params], lr=learning_rate)
params = training_loop(
    n_epochs = 5000,
    optimizer = optimizer,
    params = params,
    train_t_f = train_t_fn,
    val_t_f = val_t_fn,
    train_t_c = train_t_c,
    val_t_c = val_t_c)
```

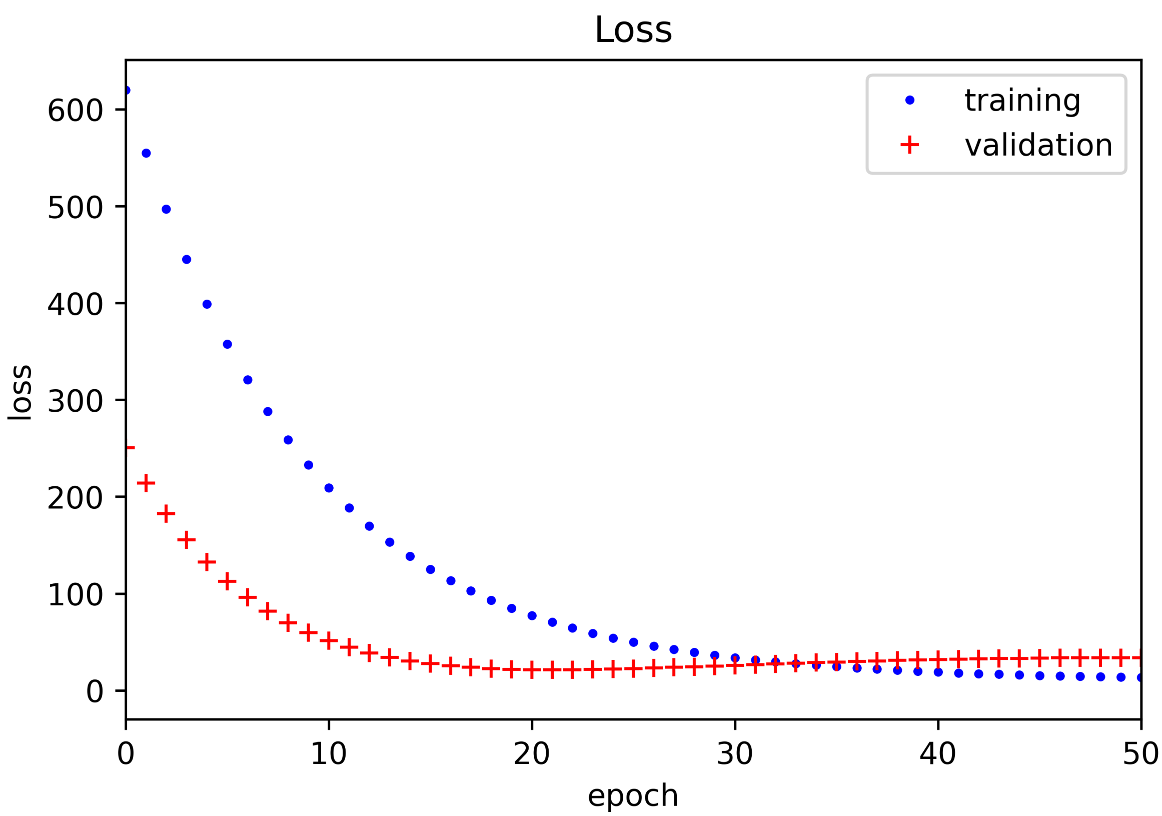
```
Epoch 1, Training loss 619.951237393134, Validation loss 250.3558618018543
Epoch 2, Training loss 554.9226239718406, Validation loss 213.950968849575
Epoch 3, Training loss 496.90544344180773, Validation loss 182.58100582534257
Epoch 500, Training loss 2.4481624373905193, Validation loss 1.19606846274614
62
Epoch 1000, Training loss 2.438438443258575, Validation loss 1.40776338911428
67
Epoch 1500, Training loss 2.438433219713113, Validation loss 1.41378842862551
07
Epoch 2000, Training loss 2.438433219713113, Validation loss 1.41378842862551
07
Epoch 2500, Training loss 2.438433219713113, Validation loss 1.41378842862551
07
Epoch 3000, Training loss 2.438433219713113, Validation loss 1.41378842862551
07
Epoch 3500, Training loss 2.438433219713113, Validation loss 1.41378842862551
07
Epoch 4000, Training loss 2.438433219713113, Validation loss 1.41378842862551
07
Epoch 4500, Training loss 2.438433219713113, Validation loss 1.41378842862551
07
Epoch 5000, Training loss 2.438433219713113, Validation loss 1.41378842862551
07
```

In [396]:
```
t_p = model(t_fn, *params[0])
# Besides the original training data, we will plot prediction `t_p` for every
 input `t_f`
fig = plt.figure(dpi=600)
plt.xlabel("serial normalized")
plt.ylabel("response")
plt.plot(t_fn.numpy(), t_p.detach().numpy(), 'd') # cannot call numpy() direct
ly, need detach()
plt.plot(t_fn.numpy(), t_c.numpy(), 'o')
plt.legend(["model","experimental"])
plt.savefig("q4plot.png", format="png")  # bookskip
```



In [0]:
```
train_loss = params[1]
val_loss = params[2]
train_loss_points = np.asarray(train_loss)
val_loss_points = np.asarray(val_loss)
```

In [426]:
```
fig = plt.figure(dpi=600)
plt.title("Loss")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.xlim(0,50)
plt.plot(train_loss_points, 'bo', markersize = 2) # cannot call numpy() directly, need detach()
plt.plot(val_loss_points, 'r+')
plt.legend(["training","validation"])
plt.savefig("q4plotloss.png", format="png")  # bookskip
```

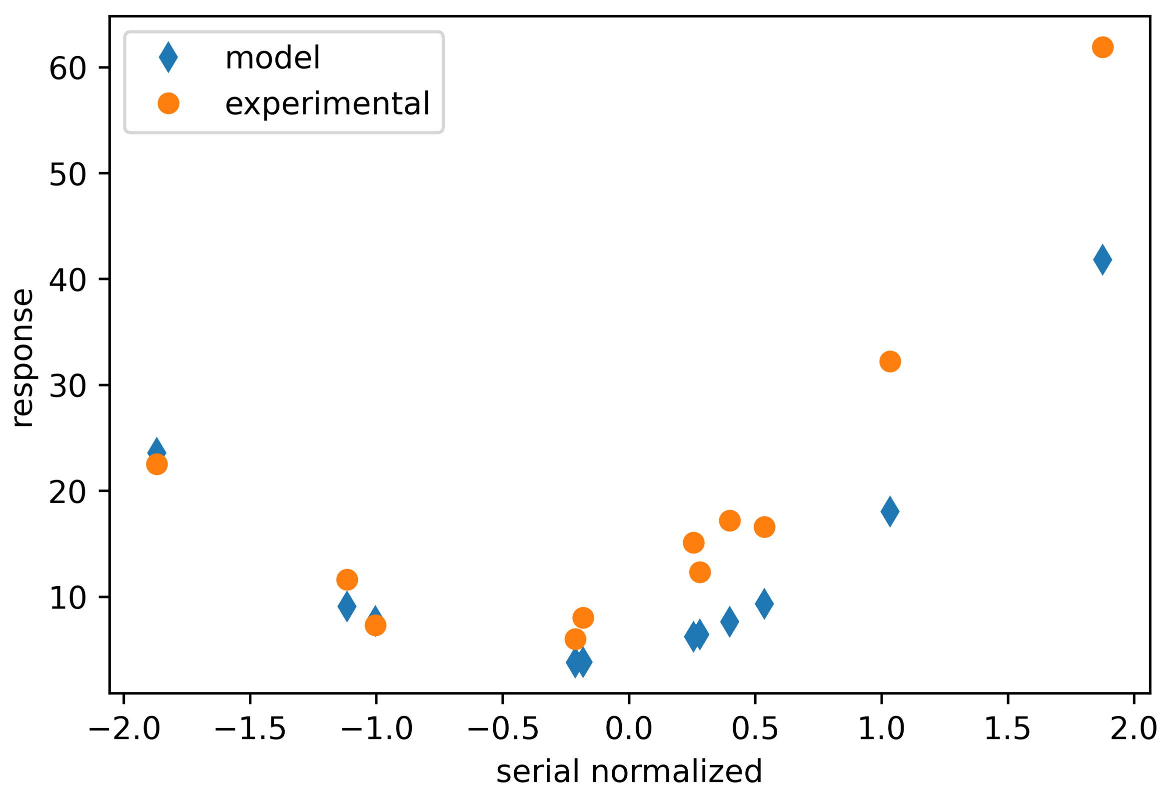


The validation loss plateaus around 17 epochs. The model is significantly overfitting. At 5000 epochs and with a tiny dataset, this was very likely to happen. This also explains why the prediction plots looked so good. I will retry running the model at 17 epochs.

In [427]:
```
params = torch.tensor([1.0, 1.0, 0.0], requires_grad=True)
learning_rate = 1e-2
optimizer = optim.SGD([params], lr=learning_rate)
params = training_loop(
    n_epochs = 17,
    optimizer = optimizer,
    params = params,
    train_t_f = train_t_fn,
    val_t_f = val_t_fn,
    train_t_c = train_t_c,
    val_t_c = val_t_c)
```

```
Epoch 1, Training loss 619.951237393134, Validation loss 250.3558618018543
Epoch 2, Training loss 554.9226239718406, Validation loss 213.950968849575
Epoch 3, Training loss 496.90544344180773, Validation loss 182.58100582534257
```

In [428]:
```
t_p = model(t_fn, *params[0])
# Besides the original training data, we will plot prediction `t_p` for every
 input `t_f`
fig = plt.figure(dpi=600)
plt.xlabel("serial normalized")
plt.ylabel("response")
plt.plot(t_fn.numpy(), t_p.detach().numpy(), 'd') # cannot call numpy() direct
ly, need detach()
plt.plot(t_fn.numpy(), t_c.numpy(), 'o')
plt.legend(["model","experimental"])
plt.savefig("q4plot.png", format="png")  # bookskip
```



In 17 epochs we still see a decent fit to the empirical data.