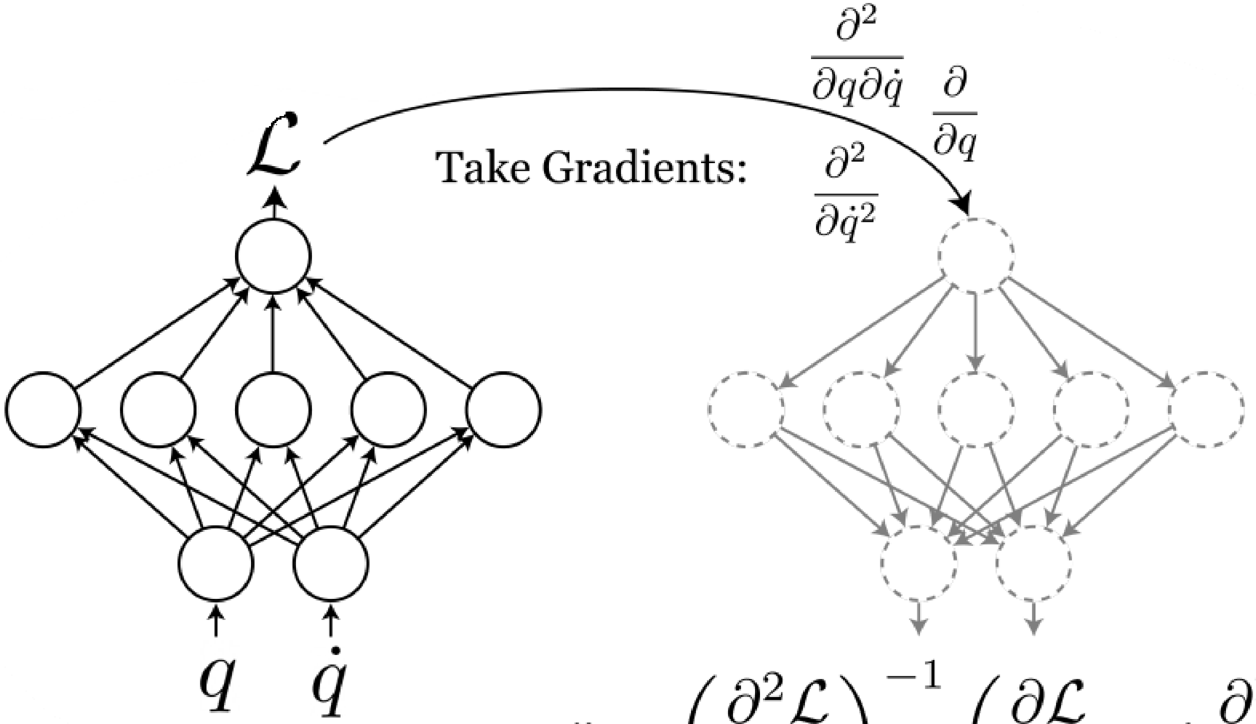
Lagrangian Neural Networks


$\mathcal{L}$
Take Gradients:
$\frac{\partial^2}{\partial q \partial \dot{q}}$
$\frac{\partial}{\partial q}$
$\frac{\partial^2}{\partial \dot{q}^2}$
q
$\dot{q}$

$$\ddot{q} = \left(\frac{\partial^2 \mathcal{L}}{\partial \dot{q}^2}\right)^{-1} \left(\frac{\partial \mathcal{L}}{\partial q} - \dot{q}\frac{\partial^2 \mathcal{L}}{\partial q \partial \dot{q}}\right)$$

Introduction

Neural Networks aren’t good at encoding symmetries without hard coded implementation:

- Hamiltonian NN can learn the conservation of energy,
  BUT require canonical coordinates $\{q_i, p_j\} = \delta_{ij}$
  (not all datasets are available in such format)

- Deep Lagrangian Networks can learn with arbitrary coordinates,
  BUT assume a quadratic kinetic term with a *q* dependant mass matrix
  (strong functional form restriction on a learnable object)

- LNN can do both with no restrictions

Euler-Lagrange equations

Using the chain rule for the time derivative in the E-L eqs.

$$\frac{d}{dt} = \dot{q}_j \frac{\partial}{\partial q_j} + \ddot{q}_j \frac{\partial}{\partial \dot{q}_j} \longrightarrow \frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{q}_i} = \frac{\partial \mathcal{L}}{\partial q_i}$$

$$\ddot{q}_j \frac{\partial^2 \mathcal{L}}{\partial \dot{q}_j \partial \dot{q}_i} + \dot{q}_j \frac{\partial^2 \mathcal{L}}{\partial q_j \partial \dot{q}_i} = \frac{\partial \mathcal{L}}{\partial q_i}$$

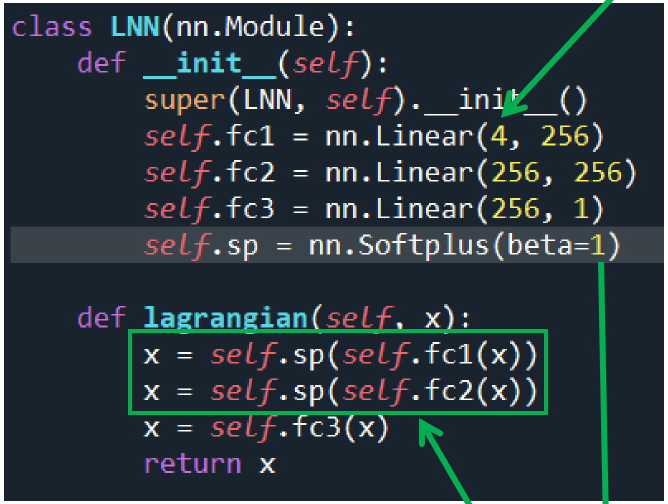
acceleration can then be computed as, in vectorial form,

$$\ddot{q} = (\nabla_{\dot{q}} \nabla_{\dot{q}}^{\top} \mathcal{L})^{-1}[\nabla_q \mathcal{L} - (\nabla_q \nabla_{\dot{q}}^{\top} \mathcal{L})\dot{q}]$$

This is what the network will predict

Structure

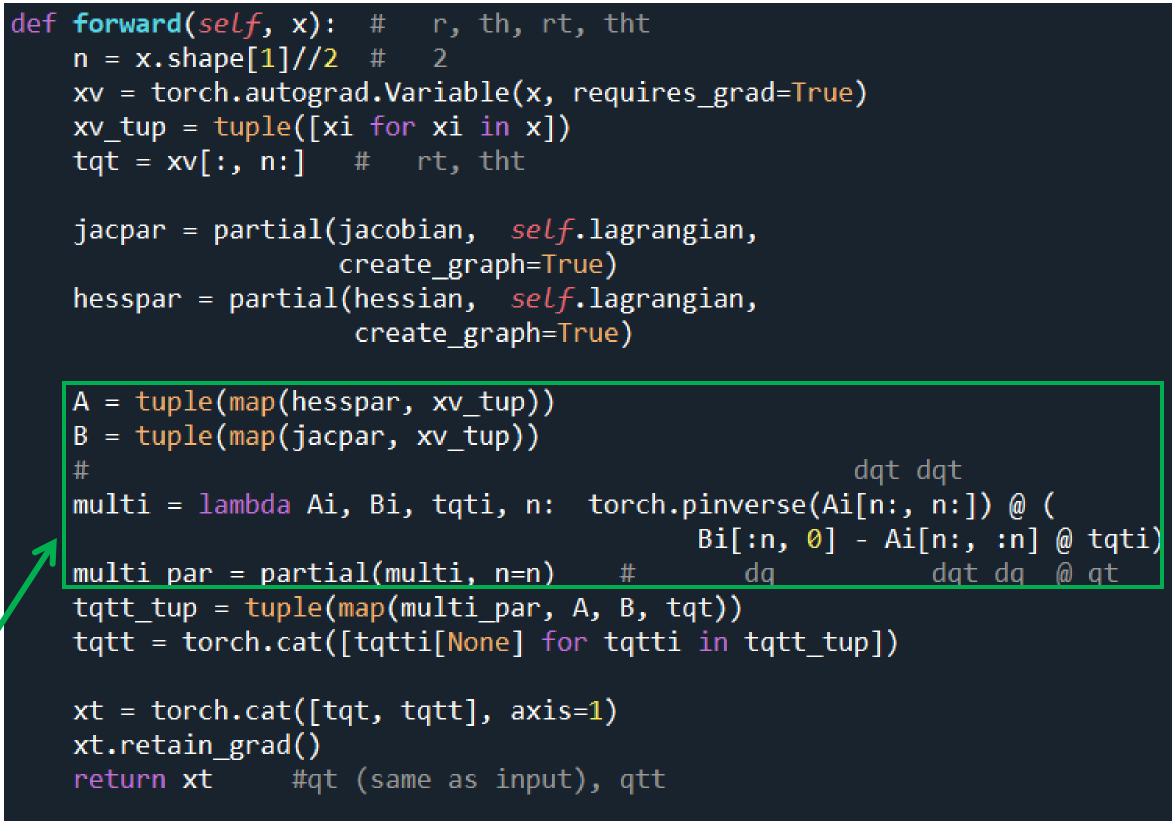
Generalised coordinates are fed to a MPL to learn a representation of L



```
class LNN(nn.Module):
    def __init__(self):
        super(LNN, self).__init__()
        self.fc1 = nn.Linear(4, 256)
        self.fc2 = nn.Linear(256, 256)
        self.fc3 = nn.Linear(256, 1)
        self.sp = nn.Softplus(beta=1)

    def lagrangian(self, x):
        x = self.sp(self.fc1(x))
        x = self.sp(self.fc2(x))
        x = self.fc3(x)
        return x
```

Softplus is used because of differential operations

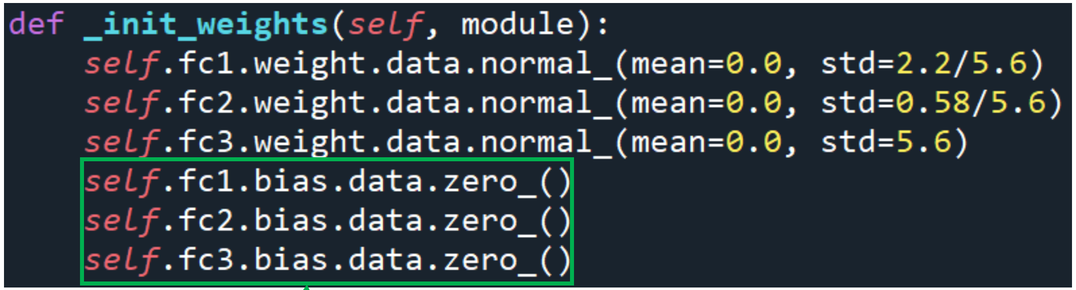
Accelerations are then computed with it



```
def forward(self, x):  #   r, th, rt, tht
    n = x.shape[1]//2  #   2
    xv = torch.autograd.Variable(x, requires_grad=True)
    xv_tup = tuple([xi for xi in x])
    tqt = xv[:, n:]   #   rt, tht

    jacpar = partial(jacobian,  self.lagrangian,
                     create_graph=True)
    hesspar = partial(hessian,  self.lagrangian,
                      create_graph=True)

    A = tuple(map(hesspar, xv_tup))
    B = tuple(map(jacpar, xv_tup))
    #                                                 dqt dqt
    multi = lambda Ai, Bi, tqti, n:  torch.pinverse(Ai[n:, n:]) @ (
                                        Bi[:n, 0] - Ai[n:, :n] @ tqti)
    multi_par = partial(multi, n=n)    #       dq           dqt dq  @ qt
    tqtt_tup = tuple(map(multi_par, A, B, tqt))
    tqtt = torch.cat([tqtti[None] for tqtti in tqtt_tup])

    xt = torch.cat([tqt, tqtt], axis=1)
    xt.retain_grad()
    return xt     #qt (same as input), qtt
```

Model initialization

Weights and biases are initialized according to the author's optimization procedure results



```
def _init_weights(self, module):
    self.fc1.weight.data.normal_(mean=0.0, std=2.2/5.6)
    self.fc2.weight.data.normal_(mean=0.0, std=0.58/5.6)
    self.fc3.weight.data.normal_(mean=0.0, std=5.6)
    self.fc1.bias.data.zero_()
    self.fc2.bias.data.zero_()
    self.fc3.bias.data.zero_()
```

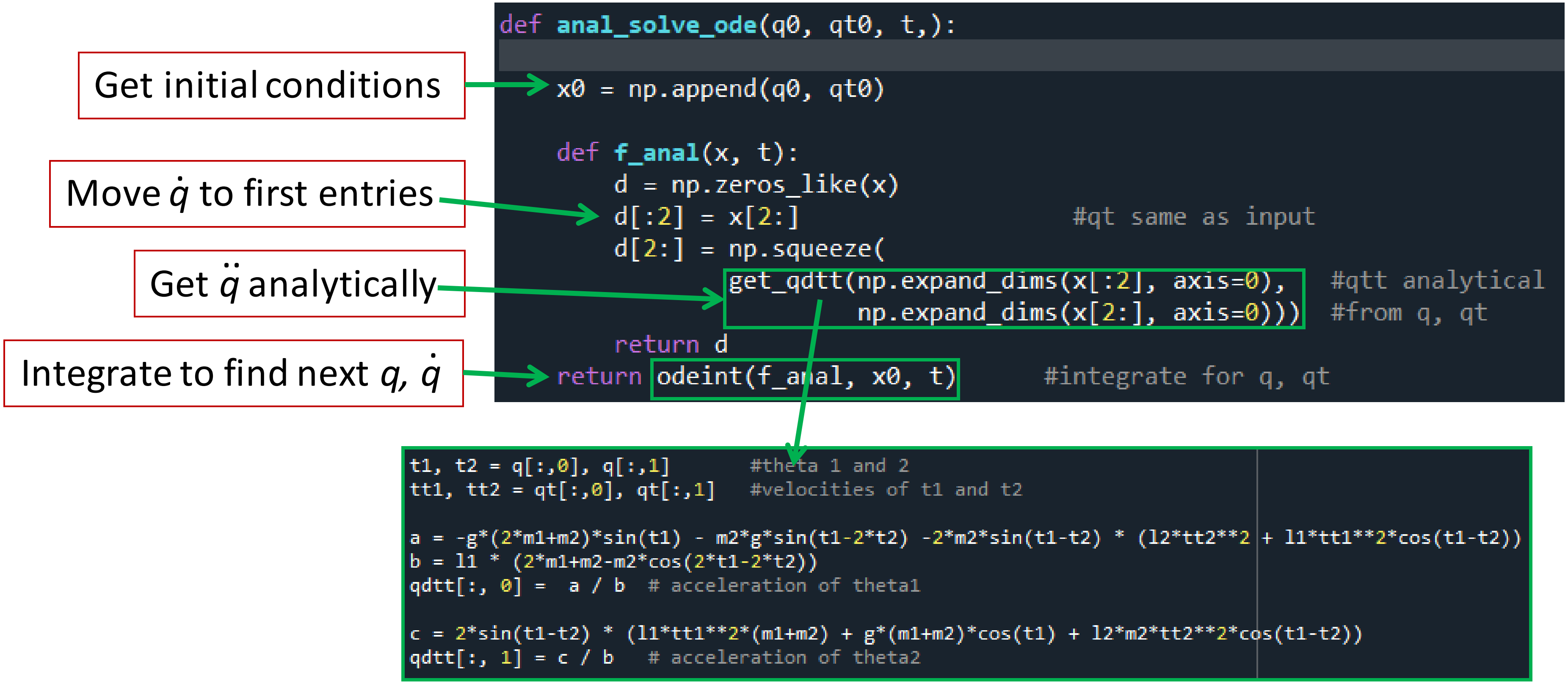
Biases are all zero

Values are picked from a Normal distribution centered at 0 with varaince depending on the layer

$$\sigma = \frac{1}{\sqrt{n}} \begin{cases} 2.2 & \text{First layer} \\ 0.58i & \text{Hidden layer } i \in \{1, \ldots\} \\ n, & \text{Output layer,} \end{cases}$$

Data

Generalised coordinates (q, $\dot{q}$) are computed analytically from a set of initial conditions. Here, ‘odeint’ from scipy.integrate is used



Get initial conditions

Move $\dot{q}$ to first entries

Get $\ddot{q}$ analytically

Integrate to find next q, $\dot{q}$

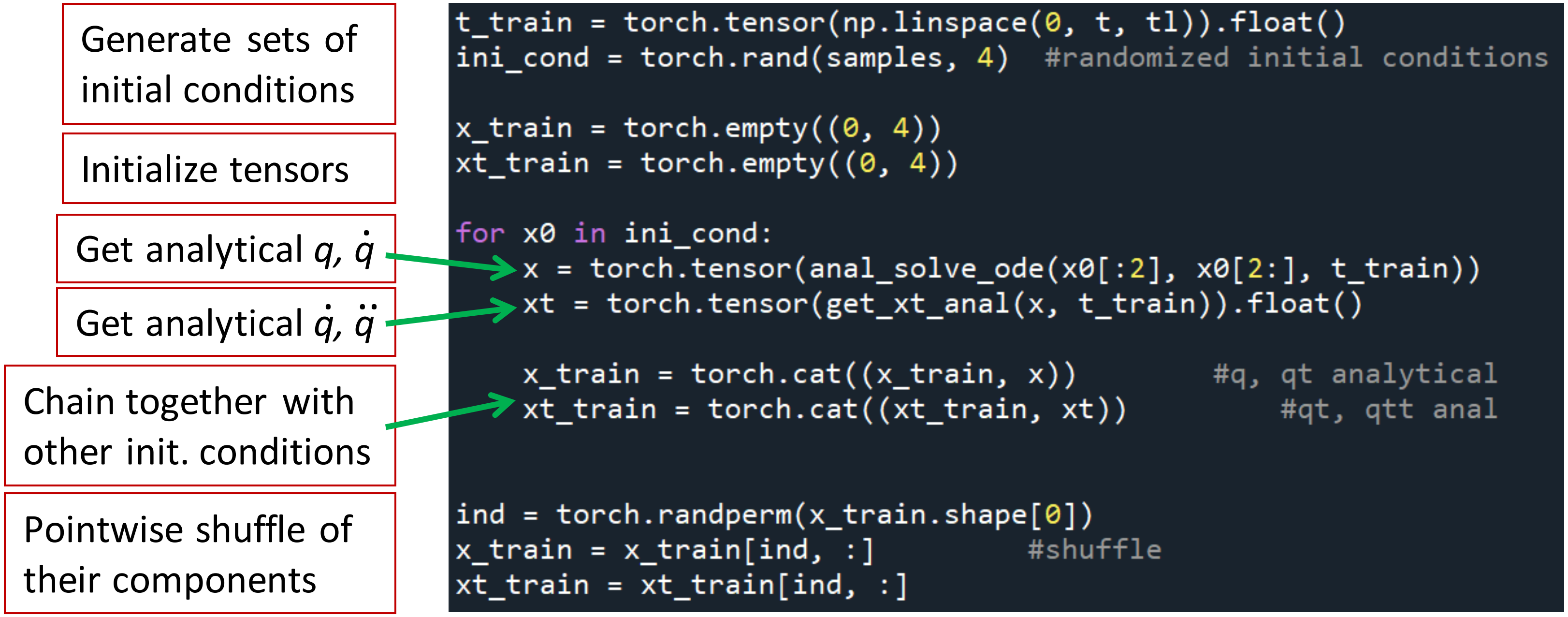
```
def anal_solve_ode(q0, qt0, t,):

    x0 = np.append(q0, qt0)

    def f_anal(x, t):
        d = np.zeros_like(x)
        d[:2] = x[2:]                    #qt same as input
        d[2:] = np.squeeze(
                    get_qdtt(np.expand_dims(x[:2], axis=0),    #qtt analytical
                             np.expand_dims(x[2:], axis=0)))   #from q, qt
        return d
    return odeint(f_anal, x0, t)       #integrate for q, qt
```

```
t1, t2 = q[:,0], q[:,1]        #theta 1 and 2
tt1, tt2 = qt[:,0], qt[:,1]    #velocities of t1 and t2

a = -g*(2*m1+m2)*sin(t1) - m2*g*sin(t1-2*t2) -2*m2*sin(t1-t2) * (l2*tt2**2 + l1*tt1**2*cos(t1-t2))
b = l1 * (2*m1+m2-m2*cos(2*t1-2*t2))
qdtt[:, 0] =  a / b  # acceleration of theta1

c = 2*sin(t1-t2) * (l1*tt1**2*(m1+m2) + g*(m1+m2)*cos(t1) + l2*m2*tt2**2*cos(t1-t2))
qdtt[:, 1] = c / b   # acceleration of theta2
```

Predicted accelerations are used instead of analytical ones to compare true and predicted coordinates

Training set

The net is trained with more than one trajectory sample



Generate sets of initial conditions

Initialize tensors

Get analytical q, $\dot{q}$

Get analytical $\dot{q}$, $\ddot{q}$

Chain together with other init. conditions

Pointwise shuffle of their components

```
t_train = torch.tensor(np.linspace(0, t, tl)).float()
ini_cond = torch.rand(samples, 4)  #randomized initial conditions

x_train = torch.empty((0, 4))
xt_train = torch.empty((0, 4))

for x0 in ini_cond:
    x = torch.tensor(anal_solve_ode(x0[:2], x0[2:], t_train))
    xt = torch.tensor(get_xt_anal(x, t_train)).float()

    x_train = torch.cat((x_train, x))        #q, qt analytical
    xt_train = torch.cat((xt_train, xt))         #qt, qtt anal


ind = torch.randperm(x_train.shape[0])
x_train = x_train[ind, :]         #shuffle
xt_train = xt_train[ind, :]
```

Model has no memory of previous points! The order in which they are fed is only relevant towards generalization power.

Training

Adam optimizer

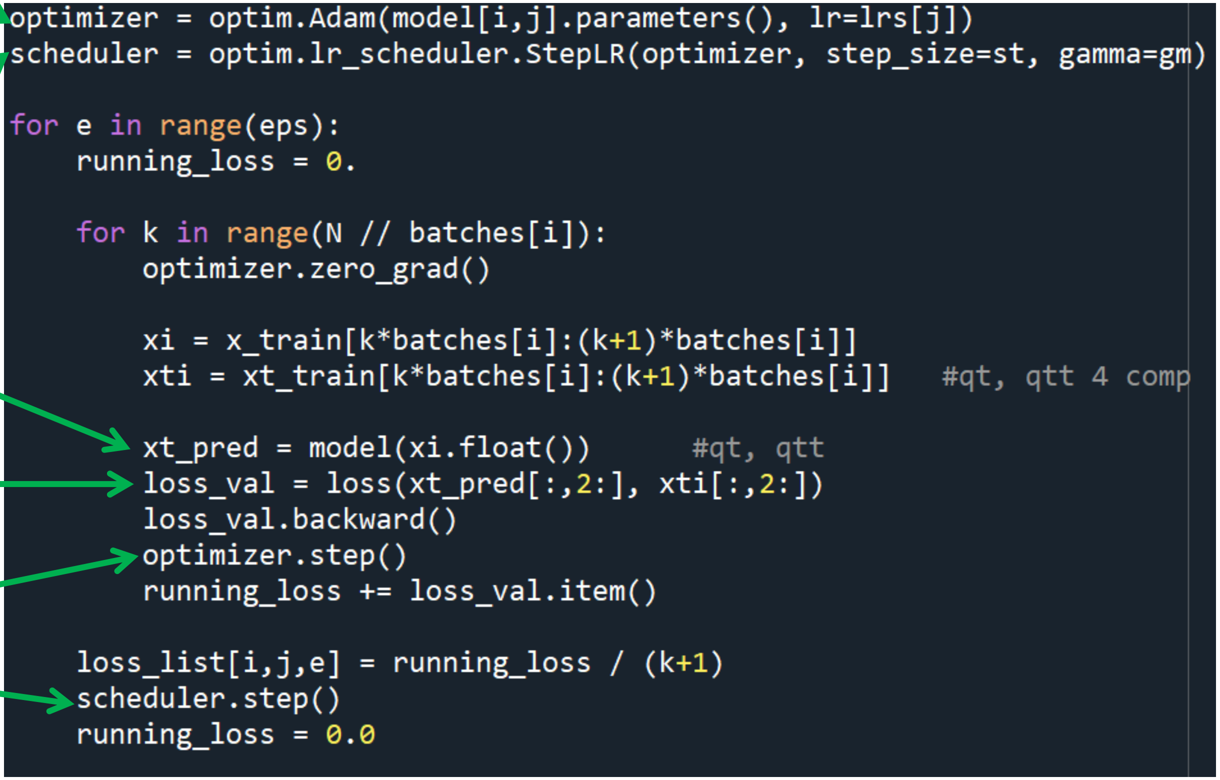
Decaying learning step:
StepLR does lr = lr*gamma
every sep_size steps

Feed (q, $\dot{q}$) to model, get $\ddot{q}$

Compute MSE between true and predicted accelerations

Backpropagation

StepLR update (if needed)

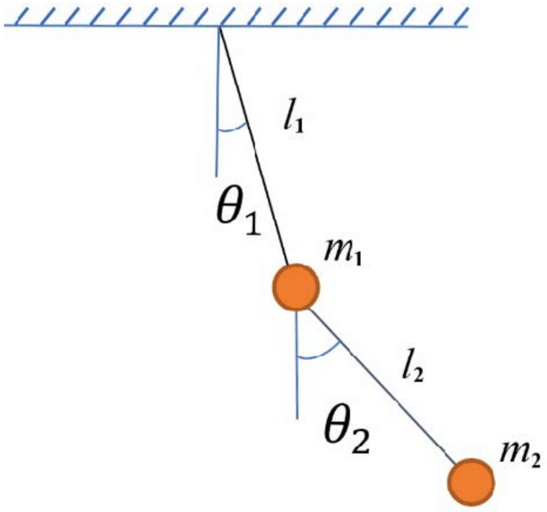


```
optimizer = optim.Adam(model[i,j].parameters(), lr=lrs[j])
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=st, gamma=gm)

for e in range(eps):
    running_loss = 0.

    for k in range(N // batches[i]):
        optimizer.zero_grad()

        xi = x_train[k*batches[i]:(k+1)*batches[i]]
        xti = xt_train[k*batches[i]:(k+1)*batches[i]]   #qt, qtt 4 comp

        xt_pred = model(xi.float())      #qt, qtt
        loss_val = loss(xt_pred[:,2:], xti[:,2:])
        loss_val.backward()
        optimizer.step()
        running_loss += loss_val.item()

    loss_list[i,j,e] = running_loss / (k+1)
    scheduler.step()
    running_loss = 0.0
```

Double Pendulum

The second system is a double pendulum

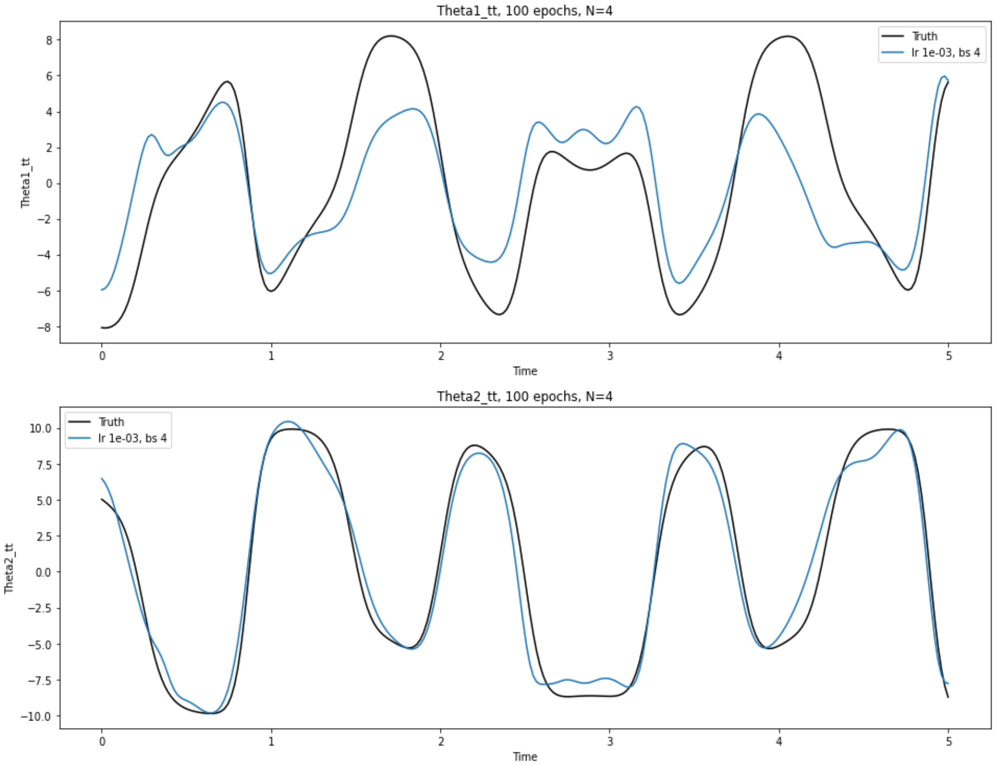
$$\mathcal{L} = \frac{1}{2}(m_1 + m_2)l_1^2\dot{\theta}_1^2 + \frac{1}{2}m_2 l_2^2\dot{\theta}_2^2 + m_2 l_1 l_2 \dot{\theta}_1 \dot{\theta}_2 \cos(\theta_1 - \theta_2) + (m_1 + m_2) g l_1 \cos\theta_1 + m_2 g l_2 \cos\theta_2$$



Masses and arm lengths are set to 1 for computational ease

Accelerations

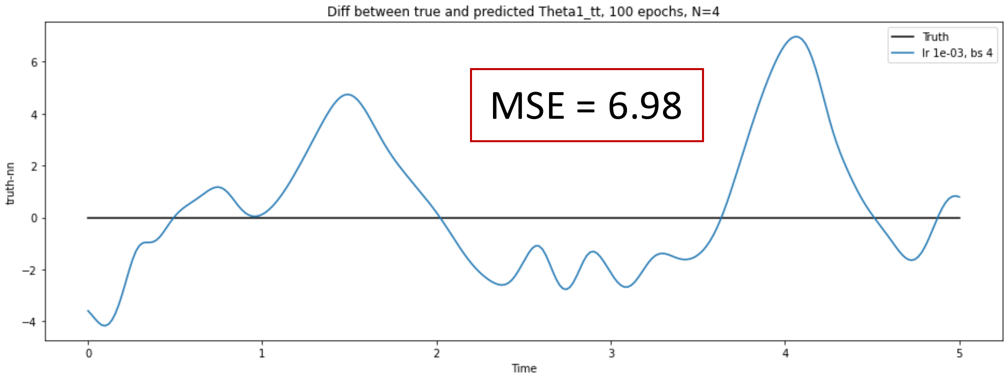
We are trying to improve TWO predictions with ONE function



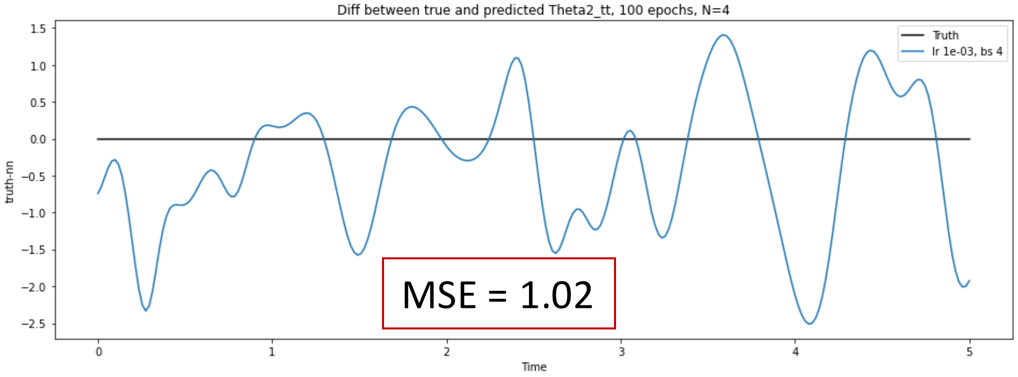
The network can find it easier to minimise the error on one component predominantly. This can be mitigated by choosing a Loss that penalises large error differences , but they would have to be non linear and slower.

Results

This is indeed the case



MSE = 6.98



MSE = 1.02

Unfortunately, it's not the end of the story.

Training is done on with small random subset of the phase space for initial conditions.

There can be configurations in which the model performs better, even on a single variable.

Results

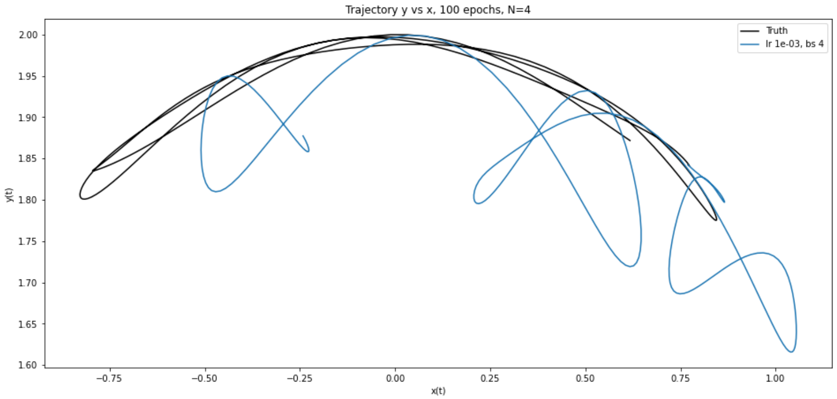
The average MSE over 1000 randomly picked trajectories is computed multiple times:

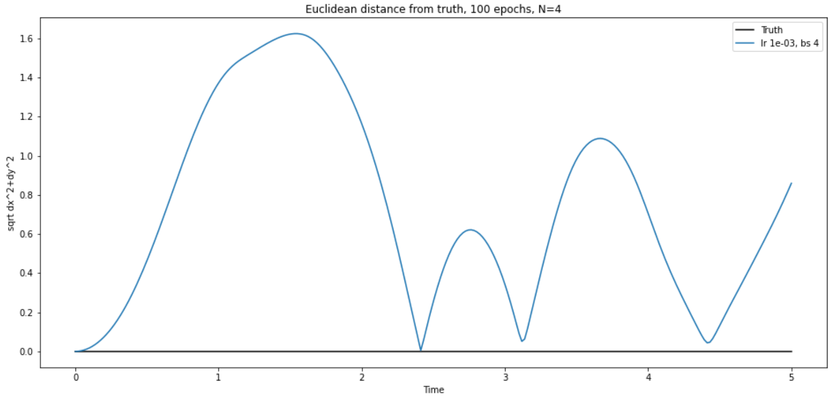
MSE $\boldsymbol{\theta}_1$	300,	0.26,	19.3,	0.63
MSE $\boldsymbol{\theta}_2$	12.8,	2.53,	127,	33,4

Model performance strongly depends on considered trajectories.
This makes it difficult to conclude if the net has a preferred coordinate.

X-Y motion

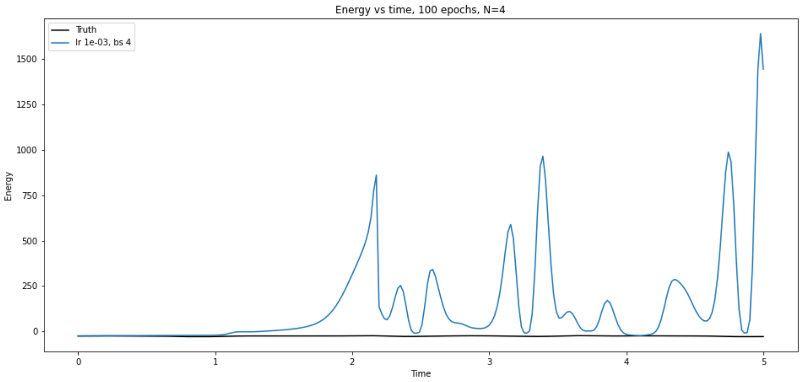
Because states are very sensitive to the previous ones, computational approximations and prediction errors can grow a lot with integration.



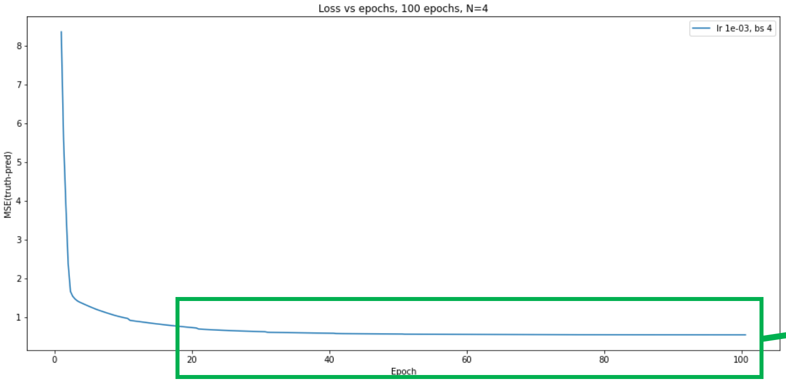


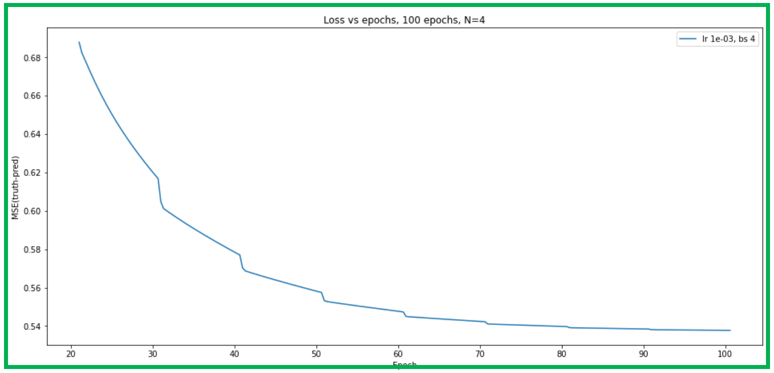
The model better predicts the trajectory for small times

Energy and Loss



Energy depends on q and $\dot{q}$, which are found by integrating predicted $\ddot{q}$. As such, the further we move from t=0, so from q_0, q_0 , the bigger the error propagation



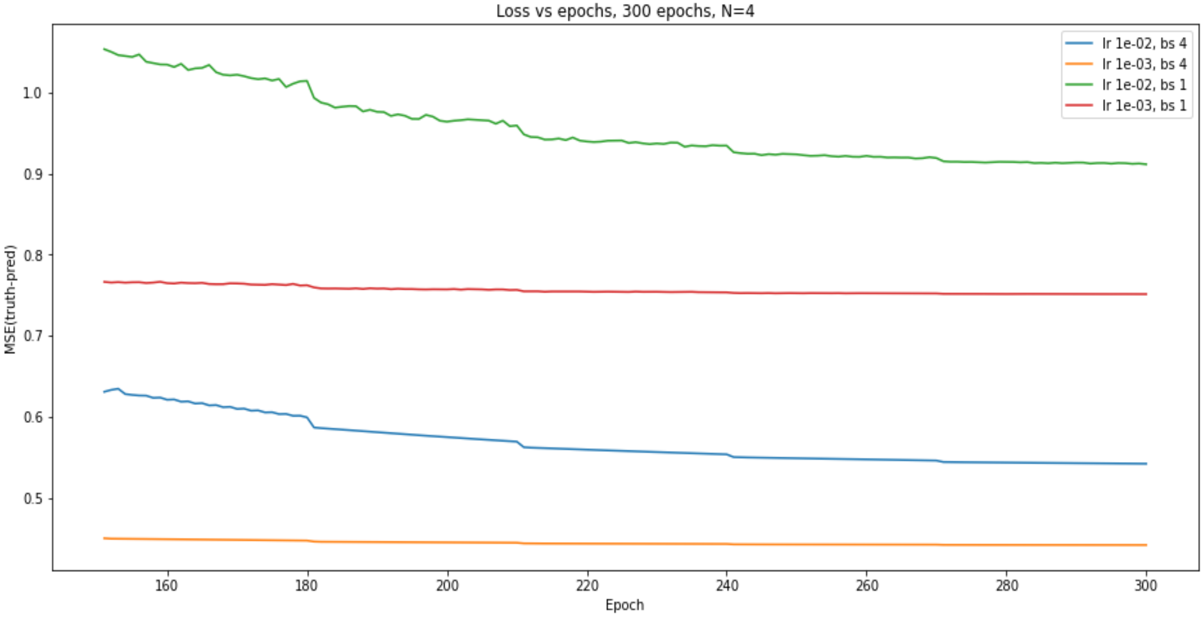


StepLR works!

With these parameters, the loss function starts plateauing within 100 epochs

Parameters sweetspot

- 3 hidden layers with 16 nodes each
- 100 epochs
- Lr_0=1e-3 x 0.5 every 10 epochs, so Lr_f=1e-6
- 4 points for each of 512 trajectories
- Batch size = 4



Spikes can appear for small batch sizes and when training with points that need much different predictions: in the case of chaotic motion, these are almost always present

Many more epochs before plateau