In this tutorial, we make an introduction to neural ordinary differential equations (NODEs) [chen2018neural]. A one-sentence summary of this model family is

an ODE system in which the differential function is a neural network (NN).

We will start this tutorial with a discussion on ODEs. Instead of presenting technique details, we will give a practical introduction to ODEs. Next, we formally describe NODEs and show three standard use cases of ODEs: classification, normalizing flows and latent dynamics learning. The lecture will be closed by works that study different aspects of the vanilla NODEs.

Organization of the Lecture

0. Introduction (10min)
1. Formal descriptions of ODEs (20min) 1.1. Computing ODE solutions 1.2. Example: Van der Pol oscillator 1.3. Break: VDP & ODE integration parameters
2. Neural ODEs (20min) 2.1. Problem formulation 2.2. Maximum likelihood estimation 2.3. Example: learning VDP sequences with NODE 2.4. Break: Adjoints
3. Latent Bayesian Neural ODEs (20min) 3.1. Variational Inference 3.2. Evidence Lowerbound 3.3. Example: Rotating MNIST 3.4. Implementation Long Break (15min)
4. ResNets are Discretized ODEs (20min) 4.1. Classification Objective 4.2. Implementation 4.3. Training 4.4. Break: ODE solver parameters
5. Continuous-time Normalizing Flows (20min) 5.1. Normalizing Flows 5.2. Continuous-time Normalizing Flows 5.3. Implementation 5.4. Training 5.5. Break: Wrap-off
6. Related Studies (15min) 6.1. ODE-RNN [rubanova2019latent] 6.2. ODE^{2}VAE [yildiz2019deep] 6.3. Augmented NODEs [dupont2019augmented] 6.4. Regularized NODEs [finlay2020train] 6.5. ACA [zhuang2020adaptive] 6.6. ODE-RL [yildiz2021continuous] 6.7. NSDEs [tzen2019neural], [xu2022infinitely] 6.8. GP-ODEs [hegde2022variational]
7. Summary & Q&A (5+25min)

NOTE: Most of the code pieces in this tutorial as well as the figures are from the original neural ODE paper and corresponding github repo.

Practicalities

- Each section ends with a 5-10-min break in which you can read the provided material and/or code snippets, ask questions, or just take a rest. Feel free to arrange your breaks in accordance with your needs.
- In addition to mathematical descriptions of the techniques, we provide short code snippets for the model definitions, training and visualization. Training could be too time consuming for this session; so make sure to load the pre-trained models if you would like to visualize the fits.
- Most of the implementation in this notebook depends on the provided utility files, some of which might be too involved to grasp immediately. If you're interested, go ahead and check them out.

The following cell imports all the required libraries.

```
%load_ext autoreload
%autoreload 2
!pip install torch torchvision torchdiffeq numpy scipy matplotlib pillow sklearn

import numpy as np
from IPython import display
import time
from sklearn.datasets import make_circles

import torch
import torch.nn as nn
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

from torchdiffeq  import odeint
from bnn          import BNN
from vae_utils    import MNIST_Encoder, MNIST_Decoder
from plot_utils   import plot_vdp_trajectories, plot_ode, plot_vdp_animation, plot_cnf_ani
    plot_mnist_sequences, plot_mnist_predictions, plot_cnf_data
from utils       import get_minibatch, mnist_loaders, inf_generator, mnist_accuracy, \
    count_parameters, conv3x3, group_norm, Flatten, load_rotating_mnist
```

1. Ordinary Differential Equations (ODEs)

Ordinary differential equations involve an independent variable, its functions and derivatives of these functions. Formally,

$$\dot{\mathbf{x}}(t) = \frac{d\mathbf{x}(t)}{dt} = \lim_{\Delta t \to 0} \frac{\mathbf{x}(t+\Delta t) - \mathbf{x}(t)}{\Delta t} = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t), \tag{1}$$

where - t denotes time (or any other independent variable) - $\mathbf{x}(t) \in \mathcal{X} \in \mathbb{R}^d$ is the state vector at time t (thus dependent variable) - $\mathbf{u}(t) \in \mathcal{A} \in \mathbb{R}^m$ is the external control signal - $\dot{\mathbf{x}}(t) \in \dot{\mathcal{X}} \in \mathbb{R}^d$ is the first order time derivative of $\mathbf{x}(t)$ - $\mathbf{f} : \mathcal{X} \times \mathcal{A} \times \mathbb{R}_+ \to \dot{\mathcal{X}}$ is the vector-valued and continuous (time) differential function describing the system's evolution over time with $\mathbb{R}_+$ denoting non-negative real numbers.

Informally speaking, $\mathbf{f}$ tells "how much the state $\mathbf{x}(t)$ would change with an infinitisemal change in t". More formally, below equation holds in the limit $\Delta t \to 0$:

$$\mathbf{x}(t+\Delta t) = \mathbf{x}(t) + \Delta t \cdot \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t). \tag{2}$$

Note-1: We often refer to $\mathbf{f}$ as *vector field* or *right hand side.* **Note-2**: Above problem is also known as *initial value problem.* **Note-3**: Throughout this tutorial, we focus on differential functions $\mathbf{f}(\mathbf{x}(t))$ independent of control signals and not explicitly parameterized by time.

1.1. Computing ODE Solutions

An "ODE state solution" $\mathbf{x}(t)$ at time $t \in \mathbb{R}_+$ is given by

$$\mathbf{x}(t) = \mathbf{x}_0 + \int_0^t \mathbf{f}(\mathbf{x}_\tau) \; d\tau, \tag{3}$$

where $\mathbf{x}_0$ denotes the initial value and $\tau \in \mathbb{R}_+$ is an auxiliary time variable.

Note-1: Given an initial value $\mathbf{x}_0$ and a set of time points $\{t_0, t_1, \dots, t_N\}$, we are often interested in state solutions $\mathbf{x}_{0:N} \equiv \{\mathbf{x}(t_0), \mathbf{x}(t_1), \dots, \mathbf{x}(t_N)\}$ **Note-2**: We occassionaly denote $\mathbf{x}_n \equiv \mathbf{x}(t_n)$. **Note-3**: Above integral has a tractable form only for very trivial differential functions (recall the integration rules from high school). Therefore, we almost always resort to numerical solvers.

Numerical solvers: TL;DR: A state solution $\mathbf{x}(t)$ can be numerically computed up to a tolerable error.

The celebrated *Picard's existence and uniqueness theorem* states that an initial value problem has a unique solution if the time differential satisfies the *Lipschitz condition.* Despite the uniqueness guarantee, there is no general recipe to analytically compute the solution; therefore, we often resort to numerical methods. The simplest and least efficient numerical method is known as *Euler's method* (above equation). More advanced methods such as *Heun's method* and *Runge-Kutta* family of solvers compute average slopes by evaluating $\mathbf{f}(\mathbf{x}(t))$ at multiple locations (speed vs accuracy trade-off). Even more advanced *adaptive step* solvers set the step size Δt dynamically.

In this tutorial, we use torchdiffeq library that implements the adjoint method for gradient estimations.

1.2. Example: Van der Pol Oscillator

As an example, we examine *Van der Pol (VDP) oscillator*, a parametric $2D$ time-invariant ODE system that evolves according to the following:

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ \mu(1-x_2^2)x_2 - x_1 \end{bmatrix}. \tag{4}$$

Our VDP implementatation below follows the two requirements of `torchdiffeq`: - Integrated function shall be an instance of `nn.Module`. - The `forward()` function must take (time,state) pair as input.

```
# define the differential function
class VDP(nn.Module):

    def __init__(self,mu):
        ''' mu is the only parameter in VDP oscillator '''
        super().__init__()
        self.mu = mu

    def forward(self, t, x):
        ''' Implements the right hand side
            Inputs
                t - []      time
                x - [N,d]   state(s)
            Output
                \dot{x} - [N,d], time derivative
        '''
        d1 = x[...,1:2]
        d2 = self.mu*(1-x[...,0:1]**2)*x[...,1:2]-x[...,0:1]
        return torch.cat([d1,d2],-1)
```

Next, we instantiate the three ingredients (differential function $\mathbf{f}$, initial value $\mathbf{x}_0$, integration time points t), forward integrate, and visualize how integration proceeds.

```
# create the differential function, needs to be a nn.Module
vdp = VDP(1.0).to(device)

# initial value, of shape [N,n]
```

```
x0 = torch.tensor([[1.0,0.0]]).float().to(device)

# integration time points, of shape [T]
ts = torch.linspace(0., 15., 500).to(device)

# forward integration
with torch.no_grad():
    X = odeint(vdp, x0, ts) # [T,N,n]

# animation
anim = plot_vdp_animation(ts,X,vdp)
display.HTML(anim.to_jshtml())
```

1.3. Break: VDP & ODE Parameters

Van der Pol oscillator has a single parameter set to 1 above: $\mu = 1$. Below cell implements the same illustration, except that we plot instead of animate. Use this break to play around with the parameter μ and initial value $\mathbf{x}_0$ to see how the tinyest change affects the whole trajectory. Note that below we visualize two trajectories as $\mathbf{x}_0$ contains two initial values.

```
# feel free to modify the parameter
vdp = VDP(5.0).to(device)

# feel free to try out different initial values
x0 = torch.tensor(
    [[-2.0,-3.0],[-2.0,3.0]]
).float().to(device)

# integration time points, of shape [T]
ts = torch.linspace(0., 15., 500).to(device)

# forward integration
with torch.no_grad():
    X = odeint(vdp, x0, ts) # [T,N,D]

plot_ode(ts,X,vdp)
```

2. Neural ODE (NODE)

To motivate the neural ODEs, imagine that we observe a sequence $\mathbf{y}_{0:N}$ that is generated by a continuous-time system (the sequence could be the measurements from a physical system, motion of objects, flow of electric current, substance rates in a chemical reaction, etc). How can we find the time evolution of such observed systems?

- If we know the underlying ODE (such as the VDP system), we can use statistics and optimization tools to estimate parameters (μ).
- What if we do not know what functional form and/or the parameters of the ODE system? We estimate the time evolution function by any function approximator (In practice, one could define GP-ODEs, linear-regression-ODEs, kernel-regression-ODEs, etc).

2.1. Problem Formulation

In more concrete terms, let's say our dataset contains a noisy observed sequence $\mathbf{y}_{0:N}$

$$\mathbf{y}_n = \mathbf{x}_n + \epsilon, \qquad \epsilon \sim \mathcal{N}(0, \sigma^2), \tag{5}$$

where each observation is a perturbation of an unknown state $\mathbf{x}_n$ generated by an unknown underlying vector field $\mathbf{f}_{\text{true}}$

$$\mathbf{x}_n = \mathbf{x}_0 + \int_0^{t_n} \mathbf{f}_{\text{true}}(\mathbf{x}_\tau)\; d\tau. \tag{6}$$

Our goal is to learn a neural network $\mathbf{f}_{\mathbf{w}}$ with parameters $\mathbf{w}$ that matches the unknown dynamics:

$$\mathbf{f}_{\mathbf{w}} \approx \mathbf{f}_{\text{true}}.$$

Let's start by implementing a NODE system. We use a simple multi-layer perceptron with two hidden layers. Since vector fields are smooth, we opt for the smooth `ELU` activation instead of `ReLU`.

```
class NODE(nn.Module):
    def __init__(self, d):
        ''' d - ODE dimensionality '''
        super().__init__()
        self._f = nn.Sequential(nn.Linear(d,200),
                                  nn.ELU(),
                                  nn.Linear(200,200),
                                  nn.ELU(),
                                  nn.Linear(200,d))

    def ode_rhs(self, t, x):
        ''' differential function = f(x)'''
        return self._f(x)

    def forward(self, ts, x0, method='dopri5'):
        ''' Forward integrates the NODE system and returns state solutions
            Input
                ts - [T]   time points
                x0 - [N,d] initial value
            Returns
                X  - [T,N,d] forward simulated states
        '''
        return odeint(self.ode_rhs, x0, ts, method=method)
```

Now, let's see what the forward trajectory $\mathbf{x}_{0:N}$ looks like when the differential function $\mathbf{f}_{\mathbf{w}}$ is a NN with randomly initialized weights $w_i \sim \mathbb{U}(-k, k)$. Here, $\mathbb{U}$ and k denote the uniform distribution and the number of input features. As you will see below, small random weights typically translate into smooth and small functions outputs, making the initial trajectory smooth as well.

```
node = NODE(2).to(device)

# let's compute the integral of our neural net!
x0 = torch.tensor([[1.0,0.0]]).float().to(device)
ts = torch.linspace(0., 20., 1000).to(device)

X = node(ts,x0)
plot_ode(ts, X, node.ode_rhs)
```

2.2. Maximum Likelihood Estimation

The simplest approach to approximate the unknown vector field $\mathbf{f}_{\text{true}}$ is the maximum-likelihood estimation. Since we do not have access to the vector field $\mathbf{f}_{\text{true}}$, we propose to match the forward simulated states with the observations:

$$\min_{\mathbf{w}} \quad \mathcal{L} = \frac{1}{2}\sum_n ||\mathbf{y}_n - \mathbf{x}_n||_2^2 \qquad \text{s.t.} \qquad \mathbf{x}_n = \mathbf{x}_0 + \int_0^{t_n} \mathbf{f}_{\mathbf{w}}(\mathbf{x}_\tau)\ d\tau. \tag{7}$$

Observe that forward simulated states $\mathbf{x}(t)$ are functions of NN parameters $\mathbf{w}$. In the following, we show the dependency explicitly by using $\mathbf{x}(t_n;\mathbf{w})$ instead of $\mathbf{x}_n$. The gradient of the loss wrt $\mathbf{w}$ can be computed by chain rule:

$$\frac{d\mathcal{L}}{d\mathbf{w}} = \sum_n (\mathbf{x}(t_n;\mathbf{w}) - \mathbf{y}_n)\frac{d\mathbf{x}(t_n;\mathbf{w})}{d\mathbf{w}} \tag{8}$$

The second term is the derivative of the forward simulated state $\frac{d\mathbf{x}(t_n;\mathbf{w})}{d\mathbf{w}}$ against the vector field parameters $\mathbf{w}$. In other words, we need to compute the derivative through the ODE solver, which is not a straightforward task. This can be done by **forward sensitivity** or **adjoints** equations. Both techniques compute the gradient by solving a second ODE system. Due to its lower memory footpring, `torchdiffeq` library implements the latter.

2.3. Example: Learning VDP Sequences with NODE

Next, we test our NODE system on noisy VDP sequences. To generate data, we randomly pick 10 initial values and forward integrate all trajectories concurrently. Luckily, this only requires setting the initial values and the rest of the implementation stays the same.

```
#| scrolled: false
# lets first generate data
vdp = VDP(1.0).to(device)
x0 = 6*torch.rand([10,2]).to(device) - 3 # 10 random initial values in [-3,3]
tvdp = torch.linspace(0., 10., 50).to(device)
with torch.no_grad():
    Xvdp = odeint(vdp, x0, tvdp)
    Yvdp = Xvdp + torch.randn_like(Xvdp)*0.1 # noisy data with observation noise has std 0

plot_vdp_trajectories(tvdp, Yvdp, vdp)
```

We now train the model on the observed sequences. To speed up the training, we optimize for a subsequence instead of the whole sequence (see `get_minibatch` function in `odevae_utils.py`).

```
#| scrolled: false
# optimization loop
Niter   = 1000 # number of optimization iterations
tsub    = 10   # subsequence length in each minibatch

optimizer = torch.optim.Adam(node.parameters(),1e-3)
for i in range(Niter):
    optimizer.zero_grad()
    t_,Y_ = get_minibatch(tvdp, Yvdp, tsub=tsub)
    Xhat = node(t_, Y_[0]) # forward simulation
    loss = ((Xhat-Y_)**2).mean() # MSE
    loss.backward()
    optimizer.step()
    if i%50==0:
        Xhat = node(tvdp, Yvdp[0]) # forward simulation
        display.clear_output(wait=True)
        plot_ode(tvdp, Yvdp, node.ode_rhs, Xhat.detach())
```

Finally, let's load and visualize a trained model.

```
state_dict = torch.load('etc/trained_node.pkl')
node.load_state_dict(state_dict)
node.eval()

Xhat = node(tvdp, Yvdp[:,0]) # forward simulation
plot_ode(tvdp, Yvdp, node.ode_rhs, Xhat.detach())
```

2.4. Break: NN Differential Function and/or Adjoints

For this break, we have two suggestions to look into: - If you would like to play around with the differential function, go ahead and try out shallower/deeper nets, other activations, smaller/larger weight initializations, etc. - If you are more into theory, take a look at the adjoints, which are the ODEs that give us the gradients of an ODE system. You can read Section 2.1 of this tutorial or Sections 1 and 3 of this techical report for a derivation of adjoints.

3. Latent Bayesian Neural ODEs (ODEVAE)

All the ODE systems we investigated so far are defined in data space, i.e., the data and the differential equation system are defined in the same space. As an example that contradicts with this modeling choice, consider the video of a flying ball. The motion of the ball can surely be explained by an ODE; however, observations themselves (pixels) do not follow any ODE at all. To handle such cases, a reasonable modeling choice is to simultaneously learn an embedding of the videos into a latent space and learn a latent ODE system that explains the motion.

A suitable generative model for a given high-dimensional observed sequence $\mathbf{y}_{0:N}$ could be as follows:

$$\mathbf{z}_0 \sim p(\mathbf{z}_0) \tag{9}$$

$$\mathbf{z}_n = \mathbf{z}_0 + \int_0^{t_n} \mathbf{f}_{\text{true}}(\mathbf{z}_\tau) d\tau \tag{10}$$

$$\mathbf{y}_n \sim p(\mathbf{y}_n | \mathbf{z}_n), \quad \forall n \in [0, N] \tag{11}$$

where $\mathbf{z}_n$ corresponds to latent embedding for $\mathbf{y}_n$. The unknowns are - the initial value for each sequence - the latent dynamics - the observation mapping.

3.1. Variational Inference

As before, we propose to infer the unknown dynamics $\mathbf{f}_{\text{true}}$ by a NODE system $\mathbf{f}_{\mathbf{w}}$. This time, our goal is to maintain uncertainty estimates over both the initial value and ODE dynamics. For this, we resort to VI with the following approximations: - amortized inference (encoder) to approximate the initial value distribution $q(\mathbf{z}_0|\mathbf{y}_{0:N})$ for an input sequence $\mathbf{y}_{0:N}$ - mean-field inference $q(\mathbf{w})$ for the dynamics parameters - a decoder $\mathbf{d}(\mathbf{z}_n)$ that gives the parameters of the observation mapping $p(\mathbf{y}_n|\mathbf{z}_n)$.

In turn, the resulting formulation becomes a hybrid ODE-VAE model. Our variational posterior factorizes as follows:

$$q(\mathbf{z}_0, \mathbf{w}|\mathbf{y}_{0:N}) = q(\mathbf{z}_0|\mathbf{y}_{0:N})\ q(\mathbf{w}),$$

where both distributions are assumed to be Gaussian with diagonal covariance. **Remark-1:** Extensions to multiple sequences would require variational posteriors for all initial values $\{\mathbf{z}_0^{(r)}\}_{r=1}^R$.

Remark-2: Our variational formulation corresponds to having a Bayesian NN differential function, i.e., BNODEs. The stocasticity of BNNs (meaning that each evaluation of a BNN on the same input would give a different output) violates ODE definition (which requires the differential function to be continuous). Therefore, our framework first draw samples from the differential function, and then uses the function draw(s) to solve ODE systems.

3.2. Evidence Lower-bound

Following the standard ELBO derivations, we end up at the following bound:

$$\log p(\mathbf{y}_{0:N}) \geq \sum_n \mathbb{E}_{q(\mathbf{z}_0,\mathbf{w}|\mathbf{y}_{0:N})}[\log p(\mathbf{y}_n|\mathbf{z}_0,\mathbf{w})] - \mathrm{KL}(q(\mathbf{z}_0|\mathbf{y}_{0:N})||p(\mathbf{z}_0)) - \mathrm{KL}(q(\mathbf{w})||p(\mathbf{w})). \tag{12}$$

Thanks to Gaussian posteriors, KL terms are tractable. The intractable expected log-likelihood is approximated by Monte Carlo sampling:

$$\mathbb{E}_{q(\mathbf{z}_0,\mathbf{w}|\mathbf{y}_{0:N})}[\log p(\mathbf{y}_{0:N}|\mathbf{z}_0,\mathbf{w})] \approx \frac{1}{L}\sum_{l=1}^{L}\sum_{n=0}^{N}\log p(\mathbf{y}_n|\mathbf{z}_0^{(l)},\mathbf{w}^{(l)}). \tag{13}$$

The following procedure specifies how to compute the likelihood given the samples $\mathbf{z}_0^{(l)}$ and $\mathbf{w}^{(l)}$:

1. Drawing an initial value and a vector field sample

$$\mathbf{z}_0^{(l)} \sim q(\mathbf{z}_0|\mathbf{y}_{0:N}) \tag{14}$$

$$\mathbf{w}^{(l)} \sim q(\mathbf{w}) \tag{15}$$

2. Forward simulating

$$\mathbf{z}_n^{(l)} = \mathbf{z}_0^{(l)} + \int_0^{t_n} \mathbf{f}_{\mathbf{w}^{(l)}}(\mathbf{z}_\tau)\ d\tau \tag{16}$$

3. Decoding

$$\mathbf{x}_n^{(l)} \equiv \mathbf{d}(\mathbf{z}_n^{(l)}), \quad \forall n \in [0, N]. \tag{17}$$

Remarks: 1. We consider a mean-field approximation for differential function parameters. 2. Initial value distribution $q(\mathbf{z}_0|\mathbf{y}_{0:N})$ is also a diagonal Gaussian whose mean and variance parameters are given by the encoder NN. 3. The ELBO is jointly optimized wrt encoder, bnode and decoder parameters.

3.3. Example Dataset: Rotating MNIST

In the following example, our dataset consists of rotating MNIST digit 3. Since each pixel value is restricted to $[0, 1]$, we opt for a Bernoulli observation model instead of Gaussian:

$$\log p(\mathbf{y}|\mathbf{x}) = \sum_n y_n \log x_n + (1 - y_n) \log(1 - x_n), \qquad \mathbf{x} = \mathbf{d}(\mathbf{z}),$$

where index n denotes the observation dimensions (not the time index).

The following cell reads the dataset.

```
# we read 1042 sequences of length 16, where each observation is a 28x28 grey-scale image
Ymnist_tr, Ymnist_test = load_rotating_mnist(device) # [T,N,1,28,28]
plot_mnist_sequences(Ymnist_tr)

# let's create artificial time points corresponding to rotation angles <===> T=16
tmnist = 0.1*torch.arange(16).to(device)
```

3.4. Implementation

We now implement our `ODEVAE` class. If you would like to learn more about the encoder and decoder implementation details, check out `vae_utils.py`.

```
from torch.distributions import Normal, kl_divergence

class ODEVAE(nn.Module):
    def __init__(self, q, n_filt=16):
        ''' Inputs:
                q      - latent dimensionaliy
                n_filt - number of filters in the first CNN layer
        '''
        super().__init__()
        self.encoder  = MNIST_Encoder(q, n_filt)
        self.bnode    = BNN(n_in=q, n_out=q, n_hid_layers=2, n_hidden=100, act='elu')
        self.decoder  = MNIST_Decoder(q, n_filt)
        self.obs_loss = nn.BCELoss(reduction='sum')
        self.q        = q

    def forward(self, ts, Y, method='dopri5'):
        ''' Performs encoding, latent forward integration and decoding.
```

```
            Note that we always draw a single sample from the encoder to improve the readi
            Inputs:
                ts - [T]           observation time points
                Y  - [T,N,1,28,28] input sequences
            Returns:
                q_z0_mu  - [N,q]           initial value means
                q_z0_sig - [N,q]           initial value std
                zt       - [T,N,q]       latent trajectoy
                Xhat     - [T,N,1,28,28] reconstructions
        '''
        [T,N,nc,d,d] = Y.shape
        # encode mean and variance
        q_z0_mu, q_z0_sig = self.encoder(Y) # N,q & N,q
        # sample differential function
        f = self.bnode.draw_f()
        ode_rhs = lambda t,x: f(x)
        # sample initial values
        z0 = q_z0_mu + q_z0_sig*torch.randn_like(q_z0_sig)
        # forward integrate
        zt = odeint(ode_rhs, z0, ts, method=method) # T,N,q
        # decode
        Xhat = self.decoder(zt) # T,N,nc,d,d
        return q_z0_mu, q_z0_sig, zt, Xhat

odevae = ODEVAE(q=8).to(device)
```

Now let's implement the ELBO.

```
def compute_elbo(odevae, ts, Y):
    ''' Computes the ELBO.
        Note that we always draw a single sample from the encoder to improve the readibili
        Inputs:
            ts - [T] observation time points
            Y  - [T,N,1,28,28] input sequences
        Returns:
            rec    - [] expected log likelihood
            kl_enc - [] the KL term due to z_0
            kl_bnn - [] the KL term due to bnn weights w
    '''
    q_z0_mu, q_z0_sig, zt, Xhat = odevae(ts, Y)
    # reconstruction
    rec = -odevae.obs_loss(Xhat,Y)
    # KL divergence on z_0
    q_z0_mu, q_z0_sig = q_z0_mu.reshape(-1), q_z0_sig.reshape(-1)
    q = Normal(q_z0_mu,q_z0_sig)
    N = Normal(torch.zeros_like(q_z0_mu),torch.ones_like(q_z0_sig))
    kl_enc = kl_divergence(q,N).sum()
    # KL divergence on bnn weights
    kl_bnn = odevae.bnode.kl()
    return rec, kl_enc, kl_bnn
```

We finally train the model.

```
#| scrolled: false
Nsub  = 25  # number of sequences in each minibatch
C     = Ymnist_tr.shape[0] / Nsub # scaling factor
Niter = 2000

optimizer = torch.optim.Adam(odevae.parameters(), 1e-3)

for i in range(Niter):
    optimizer.zero_grad()
    t_,Y_ = get_minibatch(tmnist, Ymnist_tr, Nsub=Nsub)
    rec, kl_enc, kl_bnn = compute_elbo(odevae, t_, Y_)
    rec  = rec*C
    kl   = kl_enc*C + kl_bnn
    loss = -rec + kl
    loss.backward()
    optimizer.step()
    if i%25==0:
        with torch.no_grad():
            t_,Y_ = get_minibatch(tmnist, Ymnist_tr, Nsub=5)
            q_z0_mu, q_z0_sig, zt, Xhat = odevae(t_,Y_)
            display.clear_output(wait=True)
            plot_mnist_predictions(Y_, zt, Xhat)
```

The following three cells import a trained model and then plot the training and test predictions. We first visualize PCA embeddings of the latent trajectories $\mathbf{z}_{0:N}$, where each color corresponds to the embedding of one sequence. Note that a single sample is drawn from the encoder and BNN, i.e., $L = 1$. We then visualize five sequences and corresponding predictions.

```
# load a trained model
state_dict = torch.load('etc/trained_odevae.pkl')
odevae.load_state_dict(state_dict)
odevae.eval();

t_,Y_ = get_minibatch(tmnist, Ymnist_tr, Nsub=5)
q_z0_mu, q_z0_sig, zt, Xhat = odevae(t_,Y_)
plot_mnist_predictions(Y_, zt, Xhat)
```

```
t_,Y_ = get_minibatch(tmnist, Ymnist_test, Nsub=5)
q_z0_mu, q_z0_sig, zt, Xhat = odevae(t_,Y_)
plot_mnist_predictions(Y_, zt, Xhat)
```

15-MIN BREAK

My fav online radio: https://radyobozcaada.com/player/index.html

4. ResNets are Discretized ODEs

So far, we examined NODEs from a dynamical system standpoint. We showed that NODE is an instance of ODE models in which the differential function is a neural network. Thanks to their universal approximation guarantees, NODEs can approximate any ODE system.

Our presentation is orthogonal to the original NODE paper, which describes the model starting from Residual Networks (ResNets). ResNet is among the first "very deep" networks to solve classification problems. In a nutshell, ResNets consist of layers with skip connections, leading to following transformation of the hidden state $\mathbf{x}_n$ at layer n:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{f}(\mathbf{x}_n; \theta_n),$$

where θ_n corresponds to the parameters at layer n. As we showed previously, this update equation is equivalent to computing ODE solutions with fixed time increments Δt:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta t \cdot \mathbf{f}(\mathbf{x}_n, t_n; \theta),$$

Therefore, we can interpret ResNet as a rough approximation of NODEs with fixed time increments Δt. In the following, we show how ResNets can be trivially replaced by its ODE counterpart, dubbed as "ODE Networks". Since we use adaptive step ODE solvers, which can be evaluated at any point in time, ODENets are interpreted as *infinitely deep*.