Neural Ordinary Differential Equation

SAPIENZA
UNIVERSITÀ DI ROMA

Dario Zurlo
Matteo Facci

Introduction

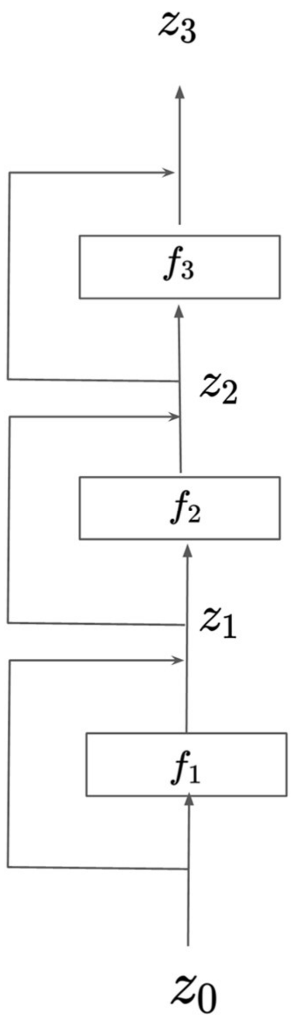
Deep Neural Networks

- ResNet
- NeuralODE

Applications

- System Identification
- ECG Heartbeat Classification

ResNet



Generic state transformation

$$z_{t+1} = z_t + f(z_t, \theta_{t+1})$$

Recursive relationship

$$z_1 = z_0 + f(z_0, \theta_1)$$

$$z_2 = z_1 + f(z_1, \theta_2)$$

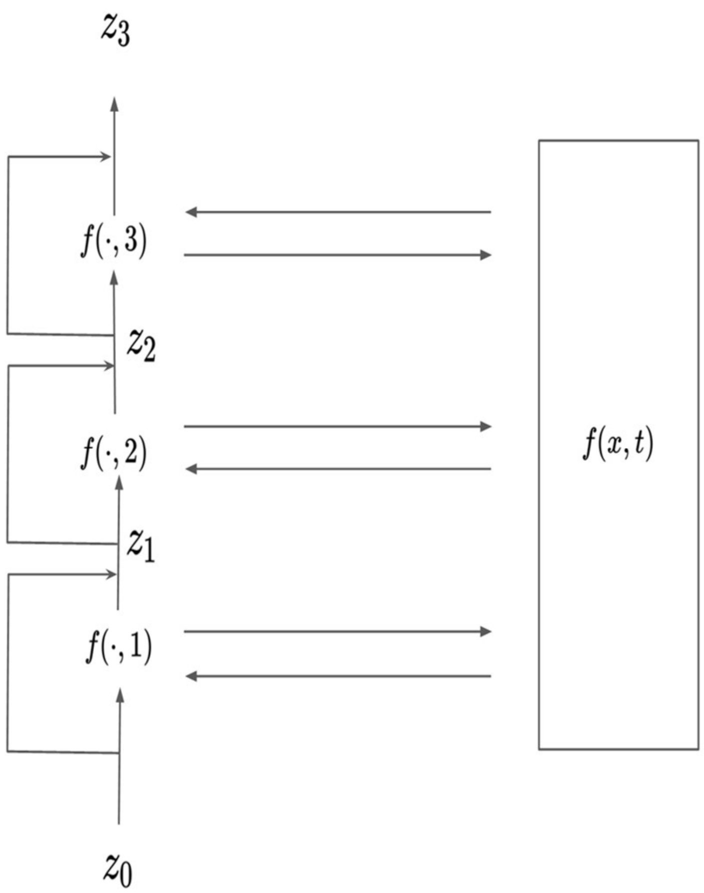
.

.

.

$$z_{t+1} = z_t + f(z_t, \theta_{t+1})$$

ODENet



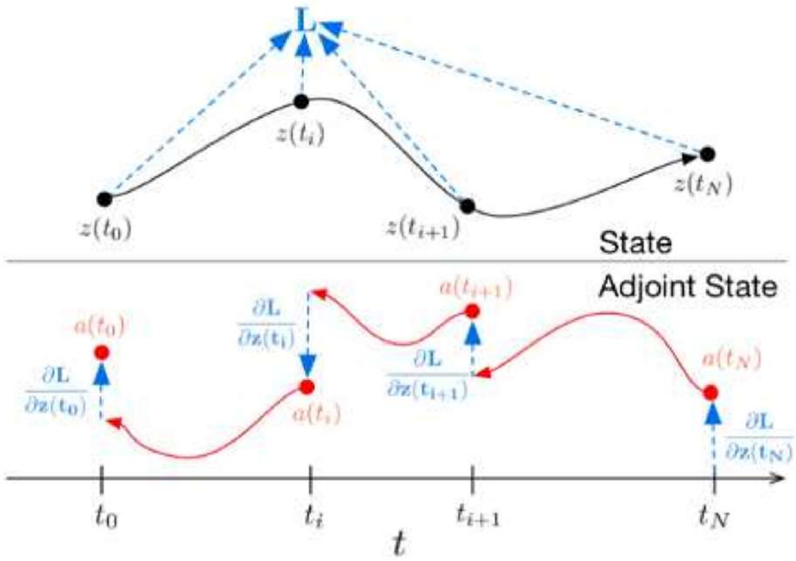
Set of ordinary differential equations

$$\frac{\partial z}{\partial t} = f(z(t), t, \theta_t)$$

Loss function

$$L(\mathbf{z}(t_1)) = L\left(\mathbf{z}(t_0) + \int_{t_0}^{t_1} f(\mathbf{z}(t), t, \theta)dt\right) = L\left(\text{ODESolve}(\mathbf{z}(t_0), f, t_0, t_1, \theta)\right)$$

Backprop Using Adjoint Method



Adjoint state

$$\mathbf{a}(t) = \frac{dL}{d\mathbf{z}(t)}$$

Time derivative of the adjoint state

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)\frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)}$$

Solution of the adjoint state

$$\underbrace{\mathbf{a}(t_N) = \frac{dL}{d\mathbf{z}(t_N)}}_{\text{initial condition of adjoint diffeq.}} \qquad \underbrace{\mathbf{a}(t_0) = \mathbf{a}(t_N) + \int_{t_N}^{t_0} \frac{d\mathbf{a}(t)}{dt}\, dt = \mathbf{a}(t_N) - \int_{t_N}^{t_0} \mathbf{a}(t)^T \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)}}_{\text{gradient wrt. initial value}}$$

Backprop Using Adjoint Method

The goal is to optimize the loss function with respect to its parameters $z(t_0), t_0, t_1, \theta$

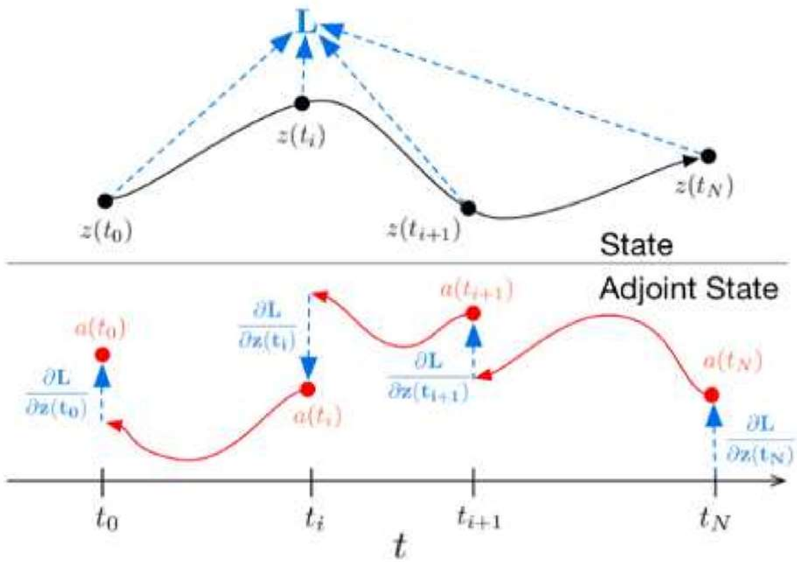
Adjoint state $a(t) = -\frac{\partial \mathcal{L}}{\partial z(t)}$

Time derivative of the adjoint state

$$\frac{da(t)}{dt} = -a(t)\frac{\partial f(z(t), t, \theta)}{\partial z}$$

Solution of the adjoint state with respect to $z(t_0)$

$$\frac{\partial \mathcal{L}}{\partial z(t_0)} = \int_{t_1}^{t_0} a(t)\frac{\partial f(z(t), t, \theta)}{\partial z} dt$$



Backprop Using Adjoint Method

Augmented state for computing all the gradients

$$\frac{d}{dt}\begin{bmatrix} z \\ \theta \\ t \end{bmatrix}(t) = f_{\text{aug}}([z, \theta, t]) := \begin{bmatrix} f([z, \theta, t]) \\ 0 \\ 1 \end{bmatrix}$$

New augmented adjoint state and gradient of the augmented dynamics

$$a_{\text{aug}} := \begin{bmatrix} a \\ a_\theta \\ a_t \end{bmatrix}, a_\theta(t) := \frac{\partial \mathcal{L}}{\partial \theta(t)}, a_t(t) := \frac{\partial \mathcal{L}}{\partial t(t)}$$

$$\frac{\partial f_{\text{aug}}}{\partial [z, \theta, t]} = \begin{bmatrix} \frac{\partial f}{\partial z} & \frac{\partial f}{\partial \theta} & \frac{\partial f}{\partial t} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

New augmented adjoint ODE

$$\frac{da_{\text{aug}}}{dt} = -\begin{bmatrix} a\frac{\partial f}{\partial z} & a\frac{\partial f}{\partial \theta} & a\frac{\partial f}{\partial t} \end{bmatrix}$$

Solutions:

$$\boxed{\begin{aligned} \frac{\partial \mathcal{L}}{\partial z(t_0)} &= \int_{t_1}^{t_0} a(t)\frac{\partial f(z(t), t, \theta)}{\partial z} dt & \frac{\partial \mathcal{L}}{\partial \theta} &= \int_{t_1}^{t_0} a(t)\frac{\partial f(z(t), t, \theta)}{\partial \theta} dt \\ \frac{\partial \mathcal{L}}{\partial t_0} &= \int_{t_1}^{t_0} a(t)\frac{\partial f(z(t), t, \theta)}{\partial t} dt & \frac{\partial \mathcal{L}}{\partial t_1} &= -a(t)\frac{\partial f(z(t), t, \theta)}{\partial t} \end{aligned}}$$

Backprop Implementation

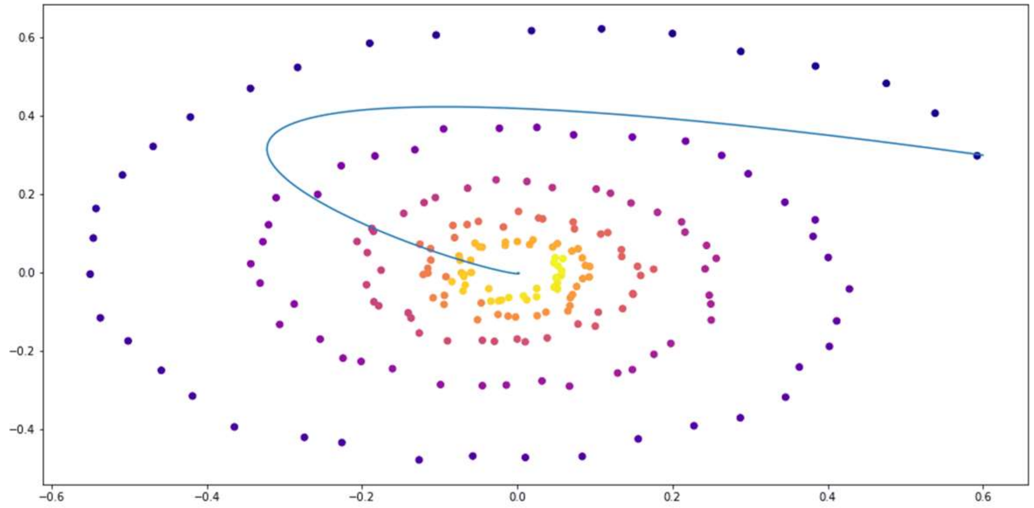
```python
class ODEAdjoint(torch.autograd.Function):
    @staticmethod
    def forward(ctx, z0, t, flat_parameters, func):
        assert isinstance(func, ODEF)
        bs, *z_shape = z0.size()
        time_len = t.size(0)

        with torch.no_grad():
            z = torch.zeros(time_len, bs, *z_shape).to(z0)
            z[0] = z0
            for i_t in range(time_len - 1):
                z0 = ode_solve(z0, t[i_t], t[i_t+1], func)
                z[i_t+1] = z0

        ctx.func = func
        ctx.save_for_backward(t, z.clone(), flat_parameters)
        return z

    @staticmethod
    def backward(ctx, dLdz):
        """
        dLdz shape: time_len, batch_size, *z_shape
        """
        func = ctx.func
        t, z, flat_parameters = ctx.saved_tensors
        time_len, bs, *z_shape = z.size()
        n_dim = np.prod(z_shape)
        n_params = flat_parameters.size(0)

        # Dynamics of augmented system to be calculated backwards in time
        def augmented_dynamics(aug_z_i, t_i):
            """
            tensors here are temporal slices
            t_i - is tensor with size: bs, 1
            aug_z_i - is tensor with size: bs, n_dim*2 + n_params + 1
            """
            z_i, a = aug_z_i[:, :n_dim], aug_z_i[:, n_dim:2*n_dim]  # ignore parameters and time

            # Unflatten z and a
            z_i = z_i.view(bs, *z_shape)
            a = a.view(bs, *z_shape)
            with torch.set_grad_enabled(True):
                t_i = t_i.detach().requires_grad_(True)
                z_i = z_i.detach().requires_grad_(True)
                func_eval, adfdz, adfdt, adfdp = func.forward_with_grad(z_i, t_i, grad_outputs=a)  # bs, *z_shape
                adfdz = adfdz.to(z_i) if adfdz is not None else torch.zeros(bs, *z_shape).to(z_i)
                adfdp = adfdp.to(z_i) if adfdp is not None else torch.zeros(bs, n_params).to(z_i)
                adfdt = adfdt.to(z_i) if adfdt is not None else torch.zeros(bs, 1).to(z_i)

            # Flatten f and adfdz
            func_eval = func_eval.view(bs, n_dim)
            adfdz = adfdz.view(bs, n_dim)
            return torch.cat((func_eval, -adfdz, -adfdp, -adfdt), dim=1)

        dLdz = dLdz.view(time_len, bs, n_dim)  # flatten dLdz for convenience
        with torch.no_grad():
            ## Create placeholders for output gradients
            # Prev computed backwards adjoints to be adjusted by direct gradients
            adj_z = torch.zeros(bs, n_dim).to(dLdz)
            adj_p = torch.zeros(bs, n_params).to(dLdz)
            # In contrast to z and p we need to return gradients for all times
            adj_t = torch.zeros(time_len, bs, 1).to(dLdz)

            for i_t in range(time_len-1, 0, -1):
                z_i = z[i_t]
                t_i = t[i_t]
                f_i = func(z_i, t_i).view(bs, n_dim)

                # Compute direct gradients
                dLdz_i = dLdz[i_t]
                dLdt_i = torch.bmm(torch.transpose(dLdz_i.unsqueeze(-1), 1, 2), f_i.unsqueeze(-1))[:, 0]

                # Adjusting adjoints with direct gradients
                adj_z += dLdz_i
                adj_t[i_t] = adj_t[i_t] - dLdt_i

                # Pack augmented variable
                aug_z = torch.cat((z_i.view(bs, n_dim), adj_z, torch.zeros(bs, n_params).to(z), adj_t[i_t]), dim=-1)

                # Solve augmented system backwards
                aug_ans = ode_solve(aug_z, t_i, t[i_t-1], augmented_dynamics)

                # Unpack solved backwards augmented system
                adj_z[:] = aug_ans[:, n_dim:2*n_dim]
                adj_p[:] += aug_ans[:, 2*n_dim:2*n_dim + n_params]
                adj_t[i_t-1] = aug_ans[:, 2*n_dim + n_params:]

                del aug_z, aug_ans

            ## Adjust 0 time adjoint with direct gradients
            # Compute direct gradients
            dLdz_0 = dLdz[0]
            dLdt_0 = torch.bmm(torch.transpose(dLdz_0.unsqueeze(-1), 1, 2), f_i.unsqueeze(-1))[:, 0]

            # Adjust adjoints
            adj_z += dLdz_0
            adj_t[0] = adj_t[0] - dLdt_0
        return adj_z.view(bs, *z_shape), adj_t, adj_p, None
```

Experiment Implementation

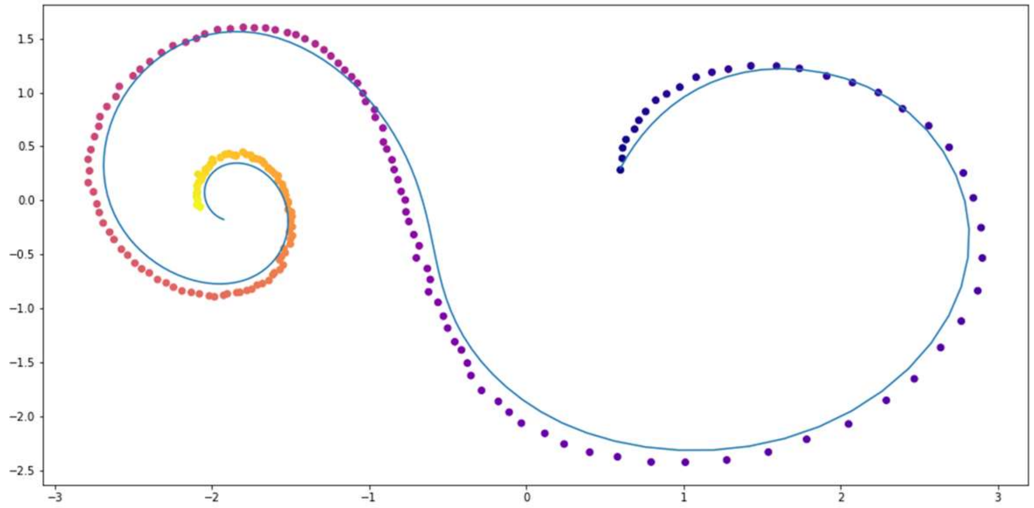
```python
def conduct_experiment(ode_true, ode_trained, n_steps, name, plot_freq=10):
    # Create data
    z0 = Variable(torch.Tensor([[0.6, 0.3]]))

    t_max = 6.29*5
    n_points = 200

    index_np = np.arange(0, n_points, 1, dtype=np.int)
    index_np = np.hstack([index_np[:, None]])
    times_np = np.linspace(0, t_max, num=n_points)
    times_np = np.hstack([times_np[:, None]])

    times = torch.from_numpy(times_np[:, :, None]).to(z0)
    obs = ode_true(z0, times, return_whole_sequence=True).detach()
    obs = obs + torch.randn_like(obs) * 0.01

    # Get trajectory of random timespan
    min_delta_time = 1.0
    max_delta_time = 5.0
    max_points_num = 32
    def create_batch():
        t0 = np.random.uniform(0, t_max - max_delta_time)
        t1 = t0 + np.random.uniform(min_delta_time, max_delta_time)

        idx = sorted(np.random.permutation(index_np[(times_np > t0) & (times_np < t1)])[:max_points_num])

        obs_ = obs[idx]
        ts_ = times[idx]
        return obs_, ts_

    # Train Neural ODE
    optimizer = torch.optim.Adam(ode_trained.parameters(), lr=0.01)
    for i in range(n_steps):
        obs_, ts_ = create_batch()

        z_ = ode_trained(obs_[0], ts_, return_whole_sequence=True)
        loss = F.mse_loss(z_, obs_.detach())

        optimizer.zero_grad()
        loss.backward(retain_graph=True)
        optimizer.step()

        if i % plot_freq == 0:
            z_p = ode_trained(z0, times, return_whole_sequence=True)

            #plot_trajectories(obs=[obs], times=[times], trajs=[z_p], save=f"{i}.png")
            plot_trajectories(obs=[obs], times=[times], trajs=[z_p])
            clear_output(wait=True)
```

System Identification Results

$$\frac{dz}{dt} = \begin{bmatrix} -0.1 & -1.0 \\ 1.0 & -0.1 \end{bmatrix} z$$



More complicated random dynamics



ECG Heartbeat Classification

- MIT-BIH ECG dataset

  110.000 annotated samples

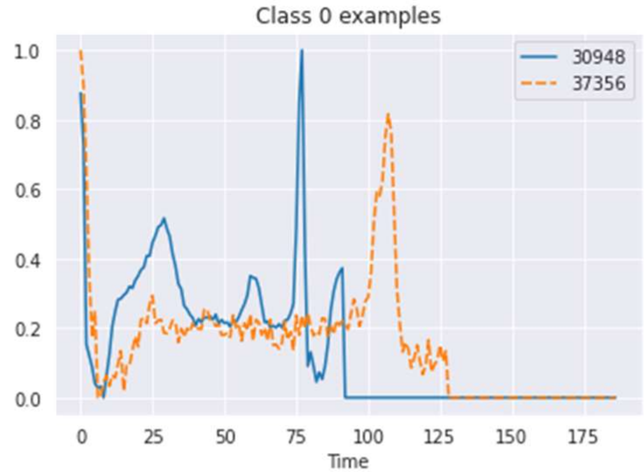
- 5 classes
  - 0: Normal beat
  - 1: Supraventricular premature beat
  - 2: Premature ventricular contraction
  - 3: Fusion of ventricular and normal beat
  - 4: Unclassified beat

Class	Training set [#]	%	Test set [#]	%
N	72471	0.828	18118	0.828
S	6431	0.073	1608	0.073
P	5788	0.066	1448	0.066
F	2223	0.025	556	0.025
U	641	0.007	162	0.007
Total	87554		21892	

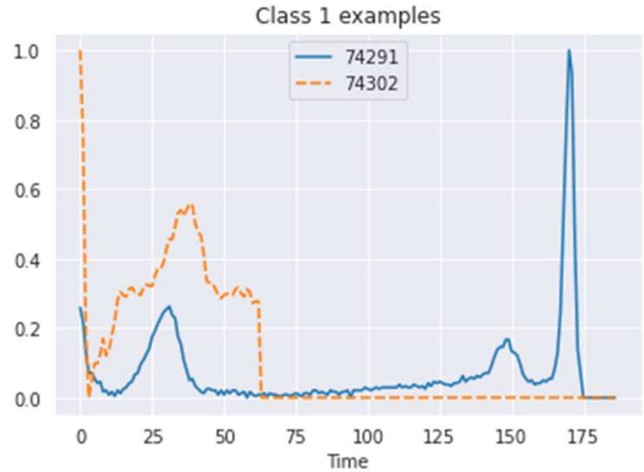
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1.000000	0.758264	0.111570	0.000000	0.080579	0.078512	0.066116	0.049587	0.047521	0.035124	0.030992	0.028926	0.035124	0.026860	0.039256
1	0.908425	0.783883	0.531136	0.362637	0.366300	0.344322	0.333333	0.307692	0.296703	0.300366	0.304029	0.336996	0.377289	0.391941	0.439560
2	0.730088	0.212389	0.000000	0.119469	0.101770	0.101770	0.110619	0.123894	0.115044	0.132743	0.106195	0.141593	0.128319	0.150442	0.132743
3	1.000000	0.910417	0.681250	0.472917	0.229167	0.068750	0.000000	0.004167	0.014583	0.054167	0.102083	0.122917	0.150000	0.168750	0.172917
4	0.570470	0.399329	0.238255	0.147651	0.000000	0.003356	0.040268	0.080537	0.070470	0.090604	0.080537	0.104027	0.093960	0.117450	0.097315

5 rows × 188 columns

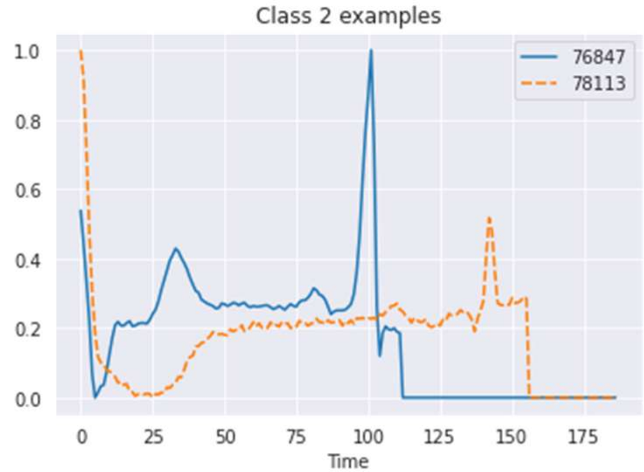
Dataset Samples



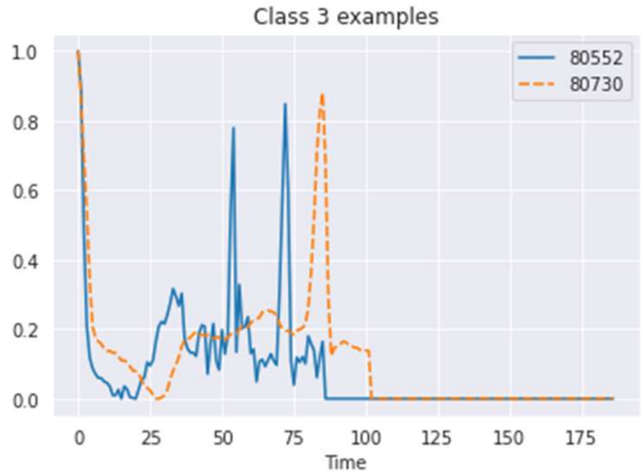
Normal beat



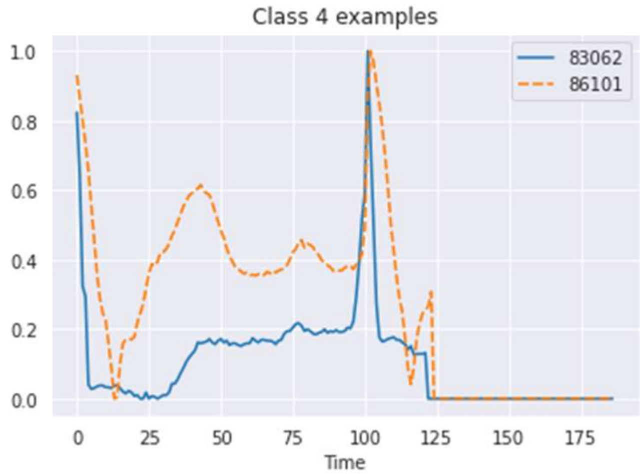
Supraventricular premature beat



Premature ventricular contraction



Fusion of ventricular and normal beat



Unclassified beat

Model Building

- ResNet feature layers:
  - Six residual blocks stacked
  - Each residual block consists of two convolutions, normalizations and ReLU activations

```
Sequential(
  (0): Conv1d(1, 64, kernel_size=(3,), stride=(1,))
  (1): GroupNorm(32, 64, eps=1e-05, affine=True)
  (2): ReLU(inplace=True)
  (3): Conv1d(64, 64, kernel_size=(4,), stride=(2,), padding=(1,))
  (4): GroupNorm(32, 64, eps=1e-05, affine=True)
  (5): ReLU(inplace=True)
  (6): Conv1d(64, 64, kernel_size=(4,), stride=(2,), padding=(1,))
  (7): ResBlock(
    (gn1): GroupNorm(32, 64, eps=1e-05, affine=True)
    (conv1): Conv1d(64, 64, kernel_size=(3,), stride=(1,), padding=(1,), bias=False)
    (gn2): GroupNorm(32, 64, eps=1e-05, affine=True)
    (conv2): Conv1d(64, 64, kernel_size=(3,), stride=(1,), padding=(1,), bias=False)
    (relu): ReLU(inplace=True)
  )
  (8): ResBlock(
    (gn1): GroupNorm(32, 64, eps=1e-05, affine=True)
    (conv1): Conv1d(64, 64, kernel_size=(3,), stride=(1,), padding=(1,), bias=False)
    (gn2): GroupNorm(32, 64, eps=1e-05, affine=True)
    (conv2): Conv1d(64, 64, kernel_size=(3,), stride=(1,), padding=(1,), bias=False)
    (relu): ReLU(inplace=True)
  )
  (9): ResBlock(
    (gn1): GroupNorm(32, 64, eps=1e-05, affine=True)
    (conv1): Conv1d(64, 64, kernel_size=(3,), stride=(1,), padding=(1,), bias=False)
    (gn2): GroupNorm(32, 64, eps=1e-05, affine=True)
    (conv2): Conv1d(64, 64, kernel_size=(3,), stride=(1,), padding=(1,), bias=False)
    (relu): ReLU(inplace=True)
  )
  (10): ResBlock(
    (gn1): GroupNorm(32, 64, eps=1e-05, affine=True)
    (conv1): Conv1d(64, 64, kernel_size=(3,), stride=(1,), padding=(1,), bias=False)
    (gn2): GroupNorm(32, 64, eps=1e-05, affine=True)
    (conv2): Conv1d(64, 64, kernel_size=(3,), stride=(1,), padding=(1,), bias=False)
    (relu): ReLU(inplace=True)
  )
  (11): ResBlock(
    (gn1): GroupNorm(32, 64, eps=1e-05, affine=True)
    (conv1): Conv1d(64, 64, kernel_size=(3,), stride=(1,), padding=(1,), bias=False)
    (gn2): GroupNorm(32, 64, eps=1e-05, affine=True)
    (conv2): Conv1d(64, 64, kernel_size=(3,), stride=(1,), padding=(1,), bias=False)
    (relu): ReLU(inplace=True)
  )
  (12): ResBlock(
    (gn1): GroupNorm(32, 64, eps=1e-05, affine=True)
    (conv1): Conv1d(64, 64, kernel_size=(3,), stride=(1,), padding=(1,), bias=False)
    (gn2): GroupNorm(32, 64, eps=1e-05, affine=True)
    (conv2): Conv1d(64, 64, kernel_size=(3,), stride=(1,), padding=(1,), bias=False)
    (relu): ReLU(inplace=True)
  )
  (13): GroupNorm(32, 64, eps=1e-05, affine=True)
  (14): ReLU(inplace=True)
  (15): AdaptiveAvgPool1d(output_size=1)
  (16): Flatten()
  (17): Linear(in_features=64, out_features=5, bias=True)
)
```

- ODENet feature layers:
  - Same structure as a single residual block
  - Uses odeint_adjoint function from torchdiffeq library:

➢ Forward: dopri5 solver
➢ Backward: adjoint method

```
Sequential(
  (0): Conv1d(1, 64, kernel_size=(3,), stride=(1,))
  (1): GroupNorm(32, 64, eps=1e-05, affine=True)
  (2): ReLU(inplace=True)
  (3): Conv1d(64, 64, kernel_size=(4,), stride=(2,), padding=(1,))
  (4): GroupNorm(32, 64, eps=1e-05, affine=True)
  (5): ReLU(inplace=True)
  (6): Conv1d(64, 64, kernel_size=(4,), stride=(2,), padding=(1,))
  (7): ODENet(
    (odefunc): ODEfunc(
      (norm1): GroupNorm(32, 64, eps=1e-05, affine=True)
      (relu): ReLU(inplace=True)
      (conv1): ConcatConv1d(
        (_layer): Conv1d(65, 64, kernel_size=(3,), stride=(1,), padding=(1,))
      )
      (norm2): GroupNorm(32, 64, eps=1e-05, affine=True)
      (conv2): ConcatConv1d(
        (_layer): Conv1d(65, 64, kernel_size=(3,), stride=(1,), padding=(1,))
      )
      (norm3): GroupNorm(32, 64, eps=1e-05, affine=True)
    )
  )
  (8): GroupNorm(32, 64, eps=1e-05, affine=True)
  (9): ReLU(inplace=True)
  (10): AdaptiveAvgPool1d(output_size=1)
  (11): Flatten()
  (12): Linear(in_features=64, out_features=5, bias=True)
)
```

ResNet vs. ODENet

- ResNet training phase:

```
Training... epoch 1
Percent trained: 100.0%  Time elapsed: 2.9 min
val loss: 0.22

Training... epoch 2
Percent trained: 100.0%  Time elapsed: 2.9 min
val loss: 0.13

Training... epoch 3
Percent trained: 100.0%  Time elapsed: 2.9 min
val loss: 0.11

Training... epoch 4
Percent trained: 100.0%  Time elapsed: 2.9 min
val loss: 0.1

Training... epoch 5
Percent trained: 100.0%  Time elapsed: 2.9 min
val loss: 0.09
```

- ODENet training phase:

```
Training... epoch 1
Percent trained: 100.0%  Time elapsed: 14.3 min
val loss: 0.23

Training... epoch 2
Percent trained: 100.0%  Time elapsed: 17.8 min
val loss: 0.16

Training... epoch 3
Percent trained: 100.0%  Time elapsed: 18.7 min
val loss: 0.12

Training... epoch 4
Percent trained: 100.0%  Time elapsed: 18.8 min
val loss: 0.11

Training... epoch 5
Percent trained: 100.0%  Time elapsed: 18.8 min
val loss: 0.12
```

Overall validation results:

```
ResNet accuracy: 0.974
ODENet accuracy: 0.969
```

```
Number of tunable parameters in...
    ResNet: 182853
    ODENet: 59333
```

Conclusions

- Pros:

- ❑ Ability to parametrize the ODE which describes the input
- ❑ Trade-off between speed and memory
- ❑ Can adjust errors
- ❑ Continuous dynamics
- ❑ Method generally applicable to numerous tasks

- Cons:

- ❑ Very slow training phase
- ❑ No advantage with respect to other architectures in case of data sampled at regular time intervals
- ❑ Issues with mini-batching
- ❑ Uniqueness of the ODE solution only if the nonlinearities be Lipschitz-continuous