demo_dkf_mvmi

December 6, 2021

1 A Deep State Space Model for Multivariate Time Series and Control Signals

1.1 Notations

- x_t: timeseries (e.g., sensor measurements)
- u_t: control signals
- z_t: latent states

1.2 References

- Kalman filter:
  https://en.wikipedia.org/wiki/Kalman_filter
- DSSM paper:
  https://dl.acm.org/doi/10.1145/3394486.3403206
- An official TensorFlow impolementation of DSSM:
  https://github.com/Google-Health/records-research/tree/master/state-space-model

[ ]:
```
import pandas as pd
import torch
import torch.nn as nn
from torch.distributions import MultivariateNormal
```

[ ]:
```
class GatedTransition(nn.Module):
    def __init__(self, z_dim, hid_dim):
        super(GatedTransition, self).__init__()

        self.gate = nn.Sequential(nn.Linear(z_dim, hid_dim),
                                  nn.ReLU(),
                                  nn.Linear(hid_dim, z_dim),
                                  nn.Sigmoid())

        self.proposed_mean = nn.Sequential(nn.Linear(z_dim, hid_dim),
                                           nn.ReLU(),
                                            nn.Linear(hid_dim, z_dim))

        self.z_to_mu = nn.Linear(z_dim, z_dim)
        # modify the default initialization of z_to_mu
        # so that it starts out as the identity function
        self.z_to_mu.weight.data = torch.eye(z_dim)
        self.z_to_mu.bias.data = torch.zeros(z_dim)

        self.z_to_logvar = nn.Linear(z_dim, z_dim)
        self.relu = nn.ReLU()

    def forward(self, z_t_1):
        #
        gate = self.gate(z_t_1)
        proposed_mean = self.proposed_mean(z_t_1)
        mu = (1 - gate) * self.z_to_mu(z_t_1) + gate * proposed_mean
        logvar = self.z_to_logvar(self.relu(proposed_mean))
        # sampling
        eps = torch.randn(z_t_1.size())
        z_t = mu + eps * torch.exp(.5 * logvar)
        return z_t, mu, logvar
```

[ ]:
```
class ControlTransition(nn.Module):
    def __init__(self, signal_dim, z_dim, hid_dim=32):
        super(ControlTransition, self).__init__()
        self.input_dim = signal_dim
        self.u_to_hidden = nn.Linear(signal_dim, hid_dim)
        self.hidden_to_z = nn.Linear(hid_dim, z_dim)

    def forward(self, u_t):
        h_t = self.u_to_hidden(u_t)
        h_t = self.hidden_to_z(h_t)
        return h_t
```

[ ]:
```
class Combiner(nn.Module):
    # PostNet
    def __init__(self, z_dim, hid_dim):
        super(Combiner, self).__init__()
        self.z_dim = z_dim
        self.z_to_hidden = nn.Linear(z_dim, hid_dim)
        self.hidden_to_mu = nn.Linear(hid_dim, z_dim)
        self.hidden_to_logvar = nn.Linear(hid_dim, z_dim)
        self.tanh = nn.Tanh()

    def forward(self, z_t_1, h_rnn):
        # combine the rnn hidden state with a transformed version of z_t_1
        # Line 577 in the original code
        h_combined = 0.5 * (self.tanh(self.z_to_hidden(z_t_1)) + h_rnn)
        # use the combined hidden state
        # to compute the mean used to sample z_t
        mu = self.hidden_to_mu(h_combined)
        # use the combined hidden state
        # to compute the scale used to sample z_t
        logvar = self.hidden_to_logvar(h_combined)
        eps = torch.randn(z_t_1.size())
        z_t = mu + eps * torch.exp(.5 * logvar)
        return z_t, mu, logvar
```

[ ]:
```
class Emitter(nn.Module):
    def __init__(self, z_dim, hid_dim, input_dim) -> None:
        super().__init__()
        self.input_dim = input_dim
        self.z_to_hidden = nn.Linear(z_dim, hid_dim)
        self.hidden_to_hidden = nn.Linear(hid_dim, hid_dim)
        self.hidden_to_input_mu = nn.Linear(hid_dim, input_dim)
        self.logvar = nn.Parameter(torch.ones(input_dim))
        self.relu = nn.ReLU()

    def forward(self, z_t):
        h1 = self.relu(self.z_to_hidden(z_t))
        h2 = self.relu(self.hidden_to_hidden(h1))
        mu = self.hidden_to_input_mu(h2)
        # return mu  # x_t
        eps = torch.randn(z_t.size(0), self.input_dim)
        x_t = mu + eps * torch.exp(.5 * self.logvar)
        return x_t, mu, self.logvar
```

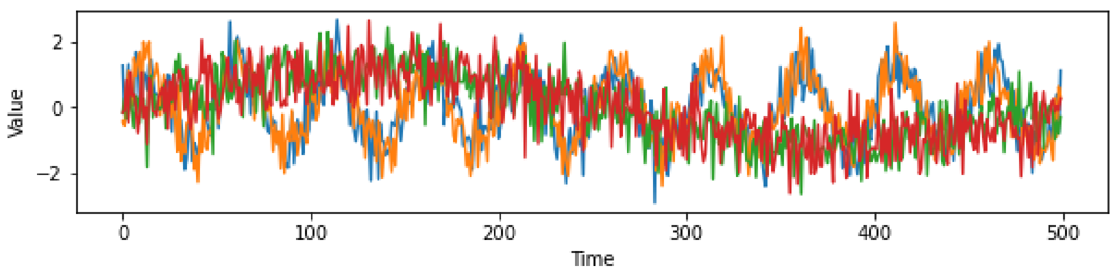
[ ]:
```
class IntervEmitter(nn.Module):
    def __init__(self, input_dim, z_dim, hid_dim=32) -> None:
        super().__init__()
        self.emitter = nn.Sequential(
            nn.Linear(z_dim, hid_dim),
            nn.Linear(hid_dim, hid_dim),
            nn.Linear(hid_dim, input_dim))

    def forward(self, z_t_1):
        return self.emitter(z_t_1)
```

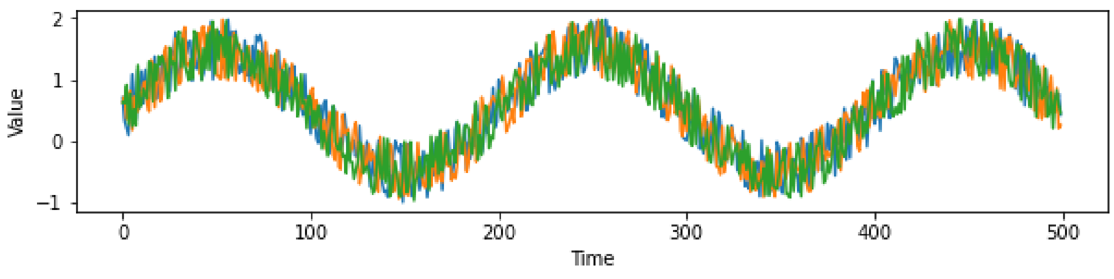
[ ]:
```
class DKF(nn.Module):
    # Structured Inference Networks
    # Current version ignores backward RNN outputs
    def __init__(self, x_dim, u_dim=0, z_dim=50,
                 trans_dim=30, emission_dim=30, interv_dim=30,
                 rnn_dim=100, num_rnn_layers=1) -> None:
```

```
        super().__init__()

        self.input_dim = x_dim
        self.signal_dim = u_dim
        self.z_dim = z_dim
        self.trans_dim = trans_dim
        self.emission_dim = emission_dim
        self.rnn_dim = rnn_dim
        self.num_rnn_layers = num_rnn_layers

        self.trans = GatedTransition(z_dim, trans_dim)
        self.emitter = Emitter(z_dim, emission_dim, x_dim)
        self.combiner = Combiner(z_dim, rnn_dim)

        if u_dim > 0:
            self.control_trans = ControlTransition(u_dim, z_dim, interv_dim)
            self.control_emitter = IntervEmitter(u_dim, z_dim, interv_dim)
            self.control_noise = nn.Parameter(torch.eye(u_dim))

        self.z_0 = nn.Parameter(torch.zeros(z_dim))
        self.z_q_0 = nn.Parameter(torch.zeros(z_dim))
        self.h_0 = nn.Parameter(torch.zeros(1, 1, rnn_dim))

        self.rnn = nn.RNN(input_size=x_dim+u_dim,
                          hidden_size=rnn_dim,
                          nonlinearity="relu",
                          batch_first=True,
                          bidirectional=False,
                          num_layers=num_rnn_layers)

    def kl_div(self, mu1, logvar1, mu2=None, logvar2=None):

        if mu2 is None:
            mu2 = torch.zeros(1, device=mu1.device)

        if logvar2 is None:
            logvar2 = torch.zeros(1, device=mu1.device)

        return torch.sum(0.5 * (
            logvar2 - logvar1 + (torch.exp(logvar1) + (mu1 - mu2).pow(2))
            / torch.exp(logvar2) - torch.ones(1, device=mu1.device)
        ), 1)

    def infer(self, x, u=None):

        batch_size, T_max, _ = x.size()
```

```python
        h_0 = self.h_0.expand(1, batch_size, self.rnn_dim).contiguous()

        if u is not None:
            # concatenate measurements and control signals
            # over their feature dimensions
            input_seq = torch.cat((x, u), -1)
        else:
            input_seq = x

        rnn_out, _ = self.rnn(input_seq, h_0)

        z_prev = self.z_q_0.expand(batch_size, self.z_q_0.size(0))
        kl_states = torch.zeros((batch_size, T_max))
        rec_losses = torch.zeros((batch_size, T_max))
        rec_u_losses = torch.zeros((batch_size, T_max))

        for t in range(T_max):
            # p(z_t|z_{t-1})
            z_prior, z_prior_mu, z_prior_logvar = self.trans(z_prev)
            if u is not None:
                # z_prior_mu is further biased by u_t
                z_prior_mu += self.control_trans(u[:, t])

            # q(z_t|z_{t-1},x_{t:T})
            z_t, z_mu, z_logvar = self.combiner(z_prev, rnn_out[:, t])
            # p(x_t|z_t)
            x_t, x_mu, x_logvar = self.emitter(z_t)

            if u is not None:
                u_mu = self.control_emitter(z_prev)
                u_dist = MultivariateNormal(
                    loc=torch.zeros(self.signal_dim),
                    scale_tril=torch.linalg.cholesky(self.control_noise))
                rec_u_losses[:, t] = -1 * u_dist.log_prob(u[:, t] - u_mu)

            # compute loss

            kl_states[:, t] = self.kl_div(
                z_mu, z_logvar, z_prior_mu, z_prior_logvar)

            rec_losses[:, t] = nn.MSELoss(reduction='none')(
                x_t.contiguous().view(-1),
                # x_mu.contiguous().view(-1),
                x[:, t].contiguous().view(-1)
            ).view(batch_size, -1).mean(dim=1)

            z_prev = z_t
        return rec_losses.mean() + rec_u_losses.mean(), kl_states.mean()

    def filter(self, x, u=None, num_sample=100):

        # Outputs
        x_hat = torch.zeros(x.size())  # predictions
        x_025 = torch.zeros(x.size())
        x_975 = torch.zeros(x.size())

        batch_size, T_max, x_dim = x.size()
        assert batch_size == 1
        z_prev = self.z_0.expand(num_sample, self.z_0.size(0))

        h_0 = self.h_0.expand(1, 1, self.rnn_dim).contiguous()
        rnn_out, _ = self.rnn(x, h_0)
        rnn_out = rnn_out.expand(num_sample,
                                 rnn_out.size(1), rnn_out.size(2))

        for t in range(T_max):
            # z_t: (num_sample, z_dim)
            z_t, z_mu, z_logvar = self.combiner(z_prev, rnn_out[:, t])
            x_t, x_mu, x_logvar = self.emitter(z_t)
            # x_hat[:, t] = x_mu

            x_covar = torch.diag(torch.sqrt(torch.exp(.5 * x_logvar)))
            x_samples = MultivariateNormal(
                x_mu, covariance_matrix=x_covar).sample()
            # # sampling z_t and computing quantiles
            # x_samples = MultivariateNormal(
            #     loc=x_mu, covariance_matrix=x_covar).sample_n(num_sample)

            x_hat[:, t] = x_samples.mean(0)
            x_025[:, t] = x_samples.quantile(0.025, 0)
            x_975[:, t] = x_samples.quantile(0.975, 0)

            # x_hat[:, t] = x_t.mean(0)
            # x_025[:, t] = x_t.quantile(0.025, 0)
            # x_975[:, t] = x_t.quantile(0.975, 0)

            z_prev = z_t
            # z_prev = z_mu

        return x_hat, x_025, x_975

    def predict(self, x, u=None, pred_steps=1, num_sample=100):
        """ x should contain the prediction period
        """
        # Outputs
        x_hat = torch.zeros(x.size())  # predictions
        x_025 = torch.zeros(x.size())
        x_975 = torch.zeros(x.size())

        if u is not None:
            u_hat = torch.zeros(u.size())  # predictions
            u_025 = torch.zeros(u.size())
            u_975 = torch.zeros(u.size())

        batch_size, T_max, x_dim = x.size()
        assert batch_size == 1
        z_prev = self.z_0.expand(num_sample, self.z_0.size(0))

        h_0 = self.h_0.expand(1, 1, self.rnn_dim).contiguous()
        if u is not None:
            # concatenate measurements and control signals
            # over their feature dimensions
            input_seq = torch.cat((x, u), -1)
        else:
            input_seq = x

        rnn_out, _ = self.rnn(input_seq[:, :T_max-pred_steps], h_0)
        rnn_out = rnn_out.expand(num_sample,
                                 rnn_out.size(1), rnn_out.size(2))

        for t in range(T_max - pred_steps):

            # z_t: (num_sample, z_dim)
            z_t, z_mu, z_logvar = self.combiner(z_prev, rnn_out[:, t])

            if u is not None:
                z_t += self.control_trans(u[:, t])
                u_mu = self.control_emitter(z_prev)
                # print(u_mu.size())  # (num_sample, z_dim)
                u_dist = MultivariateNormal(
                    loc=u_mu, scale_tril=torch.linalg.cholesky(self.
↪control_noise))

                u_samples = u_dist.sample()
                # print(u_samples.size())
                u_hat[:, t] = u_samples.mean(0)
                u_025[:, t] = u_samples.quantile(0.025, 0)
                u_975[:, t] = u_samples.quantile(0.975, 0)

            x_t, x_mu, x_logvar = self.emitter(z_t)
            x_covar = torch.diag(torch.sqrt(torch.exp(.5 * x_logvar)))
            x_samples = MultivariateNormal(
                x_mu, covariance_matrix=x_covar).sample()

            x_hat[:, t] = x_samples.mean(0)
            x_025[:, t] = x_samples.quantile(0.025, 0)
            x_975[:, t] = x_samples.quantile(0.975, 0)

            z_prev = z_mu

        for t in range(T_max - pred_steps, T_max):

            rnn_out, _ = self.rnn(input_seq[:, :t], h_0)
            rnn_out = rnn_out.expand(
                num_sample, rnn_out.size(1), rnn_out.size(2))

            z_t_1, z_mu, z_logvar = self.combiner(z_prev, rnn_out[:, -1])
            z_t, z_mu, z_logvar = self.trans(z_t_1)

            if u is not None:
                z_t += self.control_trans(u[:, t])
                u_mu = self.control_emitter(z_prev)
                # print(u_mu.size())  # (num_sample, z_dim)
                u_dist = MultivariateNormal(
                    loc=u_mu, scale_tril=torch.linalg.cholesky(self.
↪control_noise))

                u_samples = u_dist.sample()
                # print(u_samples.size())
                u_hat[:, t] = u_samples.mean(0)
                u_025[:, t] = u_samples.quantile(0.025, 0)
                u_975[:, t] = u_samples.quantile(0.975, 0)

            x_t, x_mu, x_logvar = self.emitter(z_t)

            x_covar = torch.diag(torch.sqrt(torch.exp(.5 * x_logvar)))
            x_samples = MultivariateNormal(
                x_mu, covariance_matrix=x_covar).sample()

            x_hat[:, t] = x_samples.mean(0)
            x_025[:, t] = x_samples.quantile(0.025, 0)
            x_975[:, t] = x_samples.quantile(0.975, 0)

            z_prev = z_mu

        if u is None:
            return x_hat, x_025, x_975
        else:
            return x_hat, x_025, x_975, u_hat, u_025, u_975

    def train_step(self, x, u, annealing_factor=0.1):
        self.train()
        # self.rnn.train()
        rec_loss, kl_loss = self.infer(x, u)
        total_loss = rec_loss + annealing_factor * kl_loss
        self.optimizer.zero_grad()
        total_loss.backward()
        # nn.utils.clip_grad_norm_(self.parameters(), 5.)
        self.optimizer.step()
        return rec_loss.item(), kl_loss.item(), total_loss.item()

    def validation_step(self, x, annealing_factor=0.1):
        self.eval()
        rec_loss, kl_loss = self.infer(x)
        total_loss = rec_loss + annealing_factor * kl_loss
        return rec_loss.item(), kl_loss.item(), total_loss.item()

    def fit(self, x, u, x_val=None, u_val=None,
            num_epochs=100, annealing_factor=0.1,
            verbose_step=1, eval_step=1, check_point_path=None,
            patience=20, learning_rate=0.01):

        self.optimizer = torch.optim.Adam(
            self.parameters(), lr=learning_rate)

        losses = []
        kl_losses = []
        rec_losses = []
        val_losses = []
        val_kl_losses = []
        val_rec_losses = []

        for epoch in range(num_epochs):
            try:
                res = self.train_step(x, u, annealing_factor=annealing_factor)
                losses.append(res[2])
                kl_losses.append(res[1])
                rec_losses.append(res[0])

                if epoch % verbose_step == verbose_step - 1:
                    message = f'Epoch= {epoch+1}/{num_epochs}, '
                    message += f'loss= {res[2]:.3f}, '
                    message += f'mse= {res[0]:.3f}, '
                    message += f'kld= {res[1]:.3f}'
                    print(message)

                if x_val is not None:
                    val_res = self.validation_step(x_val, annealing_factor)
                    val_losses.append(val_res[2])
                    val_kl_losses.append(val_res[1])
                    val_rec_losses.append(val_res[0])

                if epoch % eval_step == eval_step - 1 and x_val is not None:
                    message = f'\tval_loss= {val_res[2]:.3f}, '
                    message += f'val_mse= {val_res[0]:.3f}, '
                    message += f'val_kld= {val_res[1]:.3f}'
                    print(message)

            except KeyboardInterrupt:
                break

        history = {'loss': losses,
                   'kl_loss': kl_losses,
                   'rec_loss': rec_losses}

        if x_val is not None:
            history.update({'val_loss': val_losses,
                            'val_kl_loss': val_kl_losses,
                            'val_rec_loss': val_rec_losses})

        return history

    def save_model(self, filename):
        """ dkf.pth """
        torch.save(self.to('cpu').state_dict(), filename)

    def load_model(self, filename):
        self.load_state_dict(torch.load(filename))

    def get_config(self):
        return {
            'input_dim': self.input_dim,
            'z_dim': self.z_dim,
            'trans_dim': self.trans_dim,
            'emission_dim': self.emission_dim,
            'rnn_dim': self.rnn_dim,
            'num_rnn_layers': self.num_rnn_layers
        }

    @staticmethod
    def timeseries_to_tensor(x, batch_first=False):
        # output tensor shape: (batch_size, # samples, # features)
        if x.ndim == 1:
            # batch_size = 1 and # sequences = 1
            return torch.FloatTensor(x).reshape(1, len(x), 1)
        elif x.ndim == 2 and batch_first == True:
            # features = 1
            return torch.FloatTensor(x).reshape(*x.shape, 1)
        elif x.ndim == 2 and batch_first == False:
            # batch_size = 1
            return torch.FloatTensor(x).reshape(1, *x.shape)
        elif x.ndim == 3:
            return torch.FloatTensor(x)
        else:
            raise ValueError
```

[ ]:
```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.preprocessing import scale
# import warnings
# warnings.filterwarnings('ignore')

T = 500  # sequence length
observations = 2*np.sin(np.linspace(0, 20*np.pi, T))
interventions = 2*np.sin(np.linspace(0, 2*np.pi, T))

data = np.vstack([observations, observations*1.2,
                  interventions, interventions*0.85]).T
data += np.random.randn(*data.shape)
# data[:, 2:] = preprocessing.minmax_scale(data[:, 2:])
data = scale(data)

signal = np.random.rand(T, 3) + np.tile(np.sin(np.linspace(0, 5*np.pi, T)), (3,␣
 ↪1)).T

plt.figure(figsize=(10, 2))
plt.plot(data)
plt.xlabel('Time')
plt.ylabel('Value')
plt.show()

plt.figure(figsize=(10, 2))
plt.plot(signal)
plt.xlabel('Time')
plt.ylabel('Value')
plt.show()
```





[ ]:
```
dkf = DKF(data.shape[1], signal.shape[1])
```

[ ]:
```
x_ten = dkf.timeseries_to_tensor(data)
u_ten = dkf.timeseries_to_tensor(signal)
x_ten.size()
u_ten.size()
```

[ ]:
```
torch.Size([1, 500, 3])
```

[ ]:
```
history = dkf.fit(x_ten, u_ten, num_epochs=200)
history = pd.DataFrame(history)
# history
```

```
Epoch= 1/200, loss= 9.168, mse= 8.022, kld= 11.464
Epoch= 2/200, loss= 8.367, mse= 7.688, kld= 6.793
Epoch= 3/200, loss= 7.579, mse= 7.064, kld= 5.150
Epoch= 4/200, loss= 45.169, mse= 37.607, kld= 75.619
Epoch= 5/200, loss= 7.322, mse= 6.909, kld= 4.132
Epoch= 6/200, loss= 7.813, mse= 7.353, kld= 4.598
Epoch= 7/200, loss= 7.873, mse= 7.376, kld= 4.978
Epoch= 8/200, loss= 8.056, mse= 7.521, kld= 5.342
Epoch= 9/200, loss= 7.911, mse= 7.390, kld= 5.211
Epoch= 10/200, loss= 7.858, mse= 7.378, kld= 4.801
Epoch= 11/200, loss= 7.804, mse= 7.368, kld= 4.353
Epoch= 12/200, loss= 7.863, mse= 7.485, kld= 3.783
```

```
Epoch= 13/200, loss= 7.748, mse= 7.409, kld= 3.385
Epoch= 14/200, loss= 7.768, mse= 7.479, kld= 2.889
Epoch= 15/200, loss= 7.731, mse= 7.462, kld= 2.689
Epoch= 16/200, loss= 7.515, mse= 7.278, kld= 2.377
Epoch= 17/200, loss= 7.519, mse= 7.303, kld= 2.162
Epoch= 18/200, loss= 7.248, mse= 7.059, kld= 1.884
Epoch= 19/200, loss= 7.259, mse= 7.085, kld= 1.746
Epoch= 20/200, loss= 7.186, mse= 7.015, kld= 1.711
Epoch= 21/200, loss= 7.013, mse= 6.837, kld= 1.759
Epoch= 22/200, loss= 6.838, mse= 6.650, kld= 1.880
Epoch= 23/200, loss= 6.781, mse= 6.565, kld= 2.158
Epoch= 24/200, loss= 6.626, mse= 6.411, kld= 2.149
Epoch= 25/200, loss= 6.578, mse= 6.379, kld= 1.994
Epoch= 26/200, loss= 6.438, mse= 6.233, kld= 2.052
Epoch= 27/200, loss= 6.500, mse= 6.285, kld= 2.144
Epoch= 28/200, loss= 6.519, mse= 6.283, kld= 2.356
Epoch= 29/200, loss= 6.281, mse= 6.100, kld= 1.814
Epoch= 30/200, loss= 6.319, mse= 6.164, kld= 1.555
Epoch= 31/200, loss= 6.134, mse= 5.994, kld= 1.402
Epoch= 32/200, loss= 6.274, mse= 6.138, kld= 1.361
Epoch= 33/200, loss= 6.105, mse= 5.968, kld= 1.365
Epoch= 34/200, loss= 6.093, mse= 5.967, kld= 1.256
Epoch= 35/200, loss= 6.107, mse= 5.971, kld= 1.359
Epoch= 36/200, loss= 6.047, mse= 5.921, kld= 1.268
Epoch= 37/200, loss= 6.030, mse= 5.906, kld= 1.246
Epoch= 38/200, loss= 6.066, mse= 5.952, kld= 1.138
Epoch= 39/200, loss= 5.997, mse= 5.888, kld= 1.096
Epoch= 40/200, loss= 5.898, mse= 5.799, kld= 0.991
Epoch= 41/200, loss= 5.900, mse= 5.800, kld= 1.002
Epoch= 42/200, loss= 5.929, mse= 5.826, kld= 1.035
Epoch= 43/200, loss= 5.784, mse= 5.689, kld= 0.953
Epoch= 44/200, loss= 5.772, mse= 5.680, kld= 0.917
Epoch= 45/200, loss= 5.787, mse= 5.695, kld= 0.928
Epoch= 46/200, loss= 5.855, mse= 5.772, kld= 0.832
Epoch= 47/200, loss= 5.749, mse= 5.668, kld= 0.817
Epoch= 48/200, loss= 5.825, mse= 5.749, kld= 0.759
Epoch= 49/200, loss= 5.578, mse= 5.503, kld= 0.748
Epoch= 50/200, loss= 5.688, mse= 5.616, kld= 0.715
Epoch= 51/200, loss= 5.728, mse= 5.658, kld= 0.708
Epoch= 52/200, loss= 5.695, mse= 5.628, kld= 0.671
Epoch= 53/200, loss= 5.471, mse= 5.404, kld= 0.663
Epoch= 54/200, loss= 5.591, mse= 5.527, kld= 0.639
Epoch= 55/200, loss= 5.479, mse= 5.416, kld= 0.636
Epoch= 56/200, loss= 5.471, mse= 5.409, kld= 0.616
Epoch= 57/200, loss= 5.376, mse= 5.317, kld= 0.593
Epoch= 58/200, loss= 5.443, mse= 5.381, kld= 0.625
Epoch= 59/200, loss= 5.448, mse= 5.390, kld= 0.586
Epoch= 60/200, loss= 5.449, mse= 5.393, kld= 0.564
```

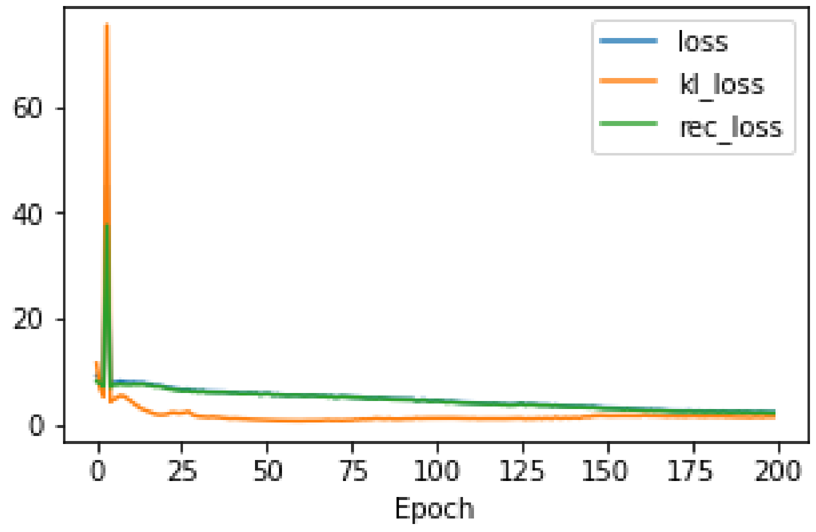
```
Epoch= 61/200, loss= 5.373, mse= 5.316, kld= 0.572
Epoch= 62/200, loss= 5.329, mse= 5.269, kld= 0.601
Epoch= 63/200, loss= 5.384, mse= 5.323, kld= 0.612
Epoch= 64/200, loss= 5.392, mse= 5.334, kld= 0.586
Epoch= 65/200, loss= 5.386, mse= 5.325, kld= 0.613
Epoch= 66/200, loss= 5.293, mse= 5.233, kld= 0.598
Epoch= 67/200, loss= 5.257, mse= 5.194, kld= 0.630
Epoch= 68/200, loss= 5.307, mse= 5.240, kld= 0.671
Epoch= 69/200, loss= 5.154, mse= 5.081, kld= 0.730
Epoch= 70/200, loss= 5.247, mse= 5.182, kld= 0.646
Epoch= 71/200, loss= 5.056, mse= 4.990, kld= 0.668
Epoch= 72/200, loss= 5.093, mse= 5.025, kld= 0.678
Epoch= 73/200, loss= 5.201, mse= 5.138, kld= 0.628
Epoch= 74/200, loss= 5.148, mse= 5.082, kld= 0.665
Epoch= 75/200, loss= 5.113, mse= 5.042, kld= 0.705
Epoch= 76/200, loss= 5.043, mse= 4.976, kld= 0.675
Epoch= 77/200, loss= 5.033, mse= 4.950, kld= 0.827
Epoch= 78/200, loss= 5.028, mse= 4.952, kld= 0.762
Epoch= 79/200, loss= 4.961, mse= 4.870, kld= 0.909
Epoch= 80/200, loss= 4.905, mse= 4.820, kld= 0.845
Epoch= 81/200, loss= 4.854, mse= 4.763, kld= 0.909
Epoch= 82/200, loss= 4.885, mse= 4.794, kld= 0.914
Epoch= 83/200, loss= 4.812, mse= 4.707, kld= 1.051
Epoch= 84/200, loss= 4.918, mse= 4.820, kld= 0.973
Epoch= 85/200, loss= 4.858, mse= 4.763, kld= 0.951
Epoch= 86/200, loss= 4.808, mse= 4.717, kld= 0.916
Epoch= 87/200, loss= 4.762, mse= 4.658, kld= 1.037
Epoch= 88/200, loss= 4.699, mse= 4.610, kld= 0.891
Epoch= 89/200, loss= 4.686, mse= 4.598, kld= 0.885
Epoch= 90/200, loss= 4.711, mse= 4.623, kld= 0.879
Epoch= 91/200, loss= 4.627, mse= 4.531, kld= 0.953
Epoch= 92/200, loss= 4.667, mse= 4.569, kld= 0.972
Epoch= 93/200, loss= 4.607, mse= 4.507, kld= 0.994
Epoch= 94/200, loss= 4.632, mse= 4.536, kld= 0.962
Epoch= 95/200, loss= 4.424, mse= 4.313, kld= 1.104
Epoch= 96/200, loss= 4.450, mse= 4.348, kld= 1.019
Epoch= 97/200, loss= 4.575, mse= 4.468, kld= 1.070
Epoch= 98/200, loss= 4.443, mse= 4.335, kld= 1.076
Epoch= 99/200, loss= 4.316, mse= 4.207, kld= 1.085
Epoch= 100/200, loss= 4.360, mse= 4.251, kld= 1.089
Epoch= 101/200, loss= 4.421, mse= 4.308, kld= 1.128
Epoch= 102/200, loss= 4.377, mse= 4.264, kld= 1.134
Epoch= 103/200, loss= 4.287, mse= 4.175, kld= 1.119
Epoch= 104/200, loss= 4.138, mse= 4.023, kld= 1.148
Epoch= 105/200, loss= 4.244, mse= 4.130, kld= 1.144
Epoch= 106/200, loss= 4.163, mse= 4.053, kld= 1.104
Epoch= 107/200, loss= 4.097, mse= 3.988, kld= 1.087
Epoch= 108/200, loss= 4.055, mse= 3.949, kld= 1.052
```

```
Epoch= 109/200, loss= 4.029, mse= 3.925, kld= 1.040
Epoch= 110/200, loss= 4.048, mse= 3.946, kld= 1.012
Epoch= 111/200, loss= 4.026, mse= 3.926, kld= 1.005
Epoch= 112/200, loss= 3.956, mse= 3.850, kld= 1.056
Epoch= 113/200, loss= 3.977, mse= 3.881, kld= 0.964
Epoch= 114/200, loss= 3.882, mse= 3.783, kld= 0.991
Epoch= 115/200, loss= 3.840, mse= 3.743, kld= 0.971
Epoch= 116/200, loss= 3.859, mse= 3.760, kld= 0.992
Epoch= 117/200, loss= 3.796, mse= 3.695, kld= 1.013
Epoch= 118/200, loss= 3.869, mse= 3.770, kld= 0.992
Epoch= 119/200, loss= 3.760, mse= 3.663, kld= 0.972
Epoch= 120/200, loss= 3.691, mse= 3.598, kld= 0.923
Epoch= 121/200, loss= 3.724, mse= 3.625, kld= 0.984
Epoch= 122/200, loss= 3.623, mse= 3.522, kld= 1.015
Epoch= 123/200, loss= 3.635, mse= 3.531, kld= 1.035
Epoch= 124/200, loss= 3.665, mse= 3.557, kld= 1.074
Epoch= 125/200, loss= 3.750, mse= 3.643, kld= 1.070
Epoch= 126/200, loss= 3.912, mse= 3.807, kld= 1.058
Epoch= 127/200, loss= 3.606, mse= 3.514, kld= 0.917
Epoch= 128/200, loss= 3.719, mse= 3.618, kld= 1.016
Epoch= 129/200, loss= 3.713, mse= 3.617, kld= 0.964
Epoch= 130/200, loss= 3.666, mse= 3.566, kld= 0.995
Epoch= 131/200, loss= 3.570, mse= 3.461, kld= 1.088
Epoch= 132/200, loss= 3.557, mse= 3.462, kld= 0.959
Epoch= 133/200, loss= 3.569, mse= 3.457, kld= 1.116
Epoch= 134/200, loss= 3.552, mse= 3.446, kld= 1.061
Epoch= 135/200, loss= 3.427, mse= 3.328, kld= 0.987
Epoch= 136/200, loss= 3.545, mse= 3.417, kld= 1.280
Epoch= 137/200, loss= 3.417, mse= 3.312, kld= 1.049
Epoch= 138/200, loss= 3.408, mse= 3.300, kld= 1.079
Epoch= 139/200, loss= 3.375, mse= 3.256, kld= 1.186
Epoch= 140/200, loss= 3.322, mse= 3.215, kld= 1.065
Epoch= 141/200, loss= 3.344, mse= 3.228, kld= 1.162
Epoch= 142/200, loss= 3.291, mse= 3.172, kld= 1.189
Epoch= 143/200, loss= 3.320, mse= 3.211, kld= 1.087
Epoch= 144/200, loss= 3.306, mse= 3.175, kld= 1.309
Epoch= 145/200, loss= 3.221, mse= 3.085, kld= 1.360
Epoch= 146/200, loss= 3.205, mse= 3.074, kld= 1.310
Epoch= 147/200, loss= 3.276, mse= 3.123, kld= 1.530
Epoch= 148/200, loss= 3.150, mse= 2.997, kld= 1.524
Epoch= 149/200, loss= 3.122, mse= 2.962, kld= 1.595
Epoch= 150/200, loss= 3.027, mse= 2.871, kld= 1.565
Epoch= 151/200, loss= 3.006, mse= 2.850, kld= 1.559
Epoch= 152/200, loss= 3.009, mse= 2.857, kld= 1.519
Epoch= 153/200, loss= 2.952, mse= 2.798, kld= 1.540
Epoch= 154/200, loss= 2.936, mse= 2.776, kld= 1.596
Epoch= 155/200, loss= 2.928, mse= 2.766, kld= 1.624
Epoch= 156/200, loss= 2.806, mse= 2.650, kld= 1.561
```

```
Epoch= 157/200, loss= 2.936, mse= 2.780, kld= 1.559
Epoch= 158/200, loss= 2.802, mse= 2.647, kld= 1.545
Epoch= 159/200, loss= 2.813, mse= 2.657, kld= 1.560
Epoch= 160/200, loss= 2.748, mse= 2.597, kld= 1.508
Epoch= 161/200, loss= 2.731, mse= 2.578, kld= 1.533
Epoch= 162/200, loss= 2.731, mse= 2.563, kld= 1.683
Epoch= 163/200, loss= 2.636, mse= 2.478, kld= 1.575
Epoch= 164/200, loss= 2.728, mse= 2.569, kld= 1.595
Epoch= 165/200, loss= 2.621, mse= 2.464, kld= 1.565
Epoch= 166/200, loss= 2.666, mse= 2.510, kld= 1.564
Epoch= 167/200, loss= 2.617, mse= 2.465, kld= 1.517
Epoch= 168/200, loss= 2.553, mse= 2.408, kld= 1.451
Epoch= 169/200, loss= 2.464, mse= 2.322, kld= 1.422
Epoch= 170/200, loss= 2.467, mse= 2.317, kld= 1.508
Epoch= 171/200, loss= 2.488, mse= 2.343, kld= 1.448
Epoch= 172/200, loss= 2.453, mse= 2.311, kld= 1.415
Epoch= 173/200, loss= 2.450, mse= 2.303, kld= 1.471
Epoch= 174/200, loss= 2.430, mse= 2.291, kld= 1.384
Epoch= 175/200, loss= 2.509, mse= 2.362, kld= 1.470
Epoch= 176/200, loss= 2.462, mse= 2.313, kld= 1.493
Epoch= 177/200, loss= 2.390, mse= 2.240, kld= 1.502
Epoch= 178/200, loss= 2.461, mse= 2.320, kld= 1.408
Epoch= 179/200, loss= 2.423, mse= 2.280, kld= 1.434
Epoch= 180/200, loss= 2.352, mse= 2.214, kld= 1.383
Epoch= 181/200, loss= 2.423, mse= 2.276, kld= 1.464
Epoch= 182/200, loss= 2.415, mse= 2.271, kld= 1.440
Epoch= 183/200, loss= 2.394, mse= 2.255, kld= 1.395
Epoch= 184/200, loss= 2.310, mse= 2.168, kld= 1.418
Epoch= 185/200, loss= 2.430, mse= 2.281, kld= 1.485
Epoch= 186/200, loss= 2.356, mse= 2.219, kld= 1.365
Epoch= 187/200, loss= 2.305, mse= 2.166, kld= 1.395
Epoch= 188/200, loss= 2.313, mse= 2.178, kld= 1.346
Epoch= 189/200, loss= 2.315, mse= 2.182, kld= 1.327
Epoch= 190/200, loss= 2.384, mse= 2.251, kld= 1.332
Epoch= 191/200, loss= 2.255, mse= 2.124, kld= 1.307
Epoch= 192/200, loss= 2.342, mse= 2.211, kld= 1.310
Epoch= 193/200, loss= 2.268, mse= 2.142, kld= 1.258
Epoch= 194/200, loss= 2.244, mse= 2.111, kld= 1.327
Epoch= 195/200, loss= 2.222, mse= 2.086, kld= 1.360
Epoch= 196/200, loss= 2.297, mse= 2.164, kld= 1.331
Epoch= 197/200, loss= 2.256, mse= 2.128, kld= 1.277
Epoch= 198/200, loss= 2.206, mse= 2.066, kld= 1.401
Epoch= 199/200, loss= 2.209, mse= 2.074, kld= 1.352
Epoch= 200/200, loss= 2.212, mse= 2.075, kld= 1.372
```

[ ]:
```
history.plot(figsize=(5, 3), xlabel='Epoch')
```

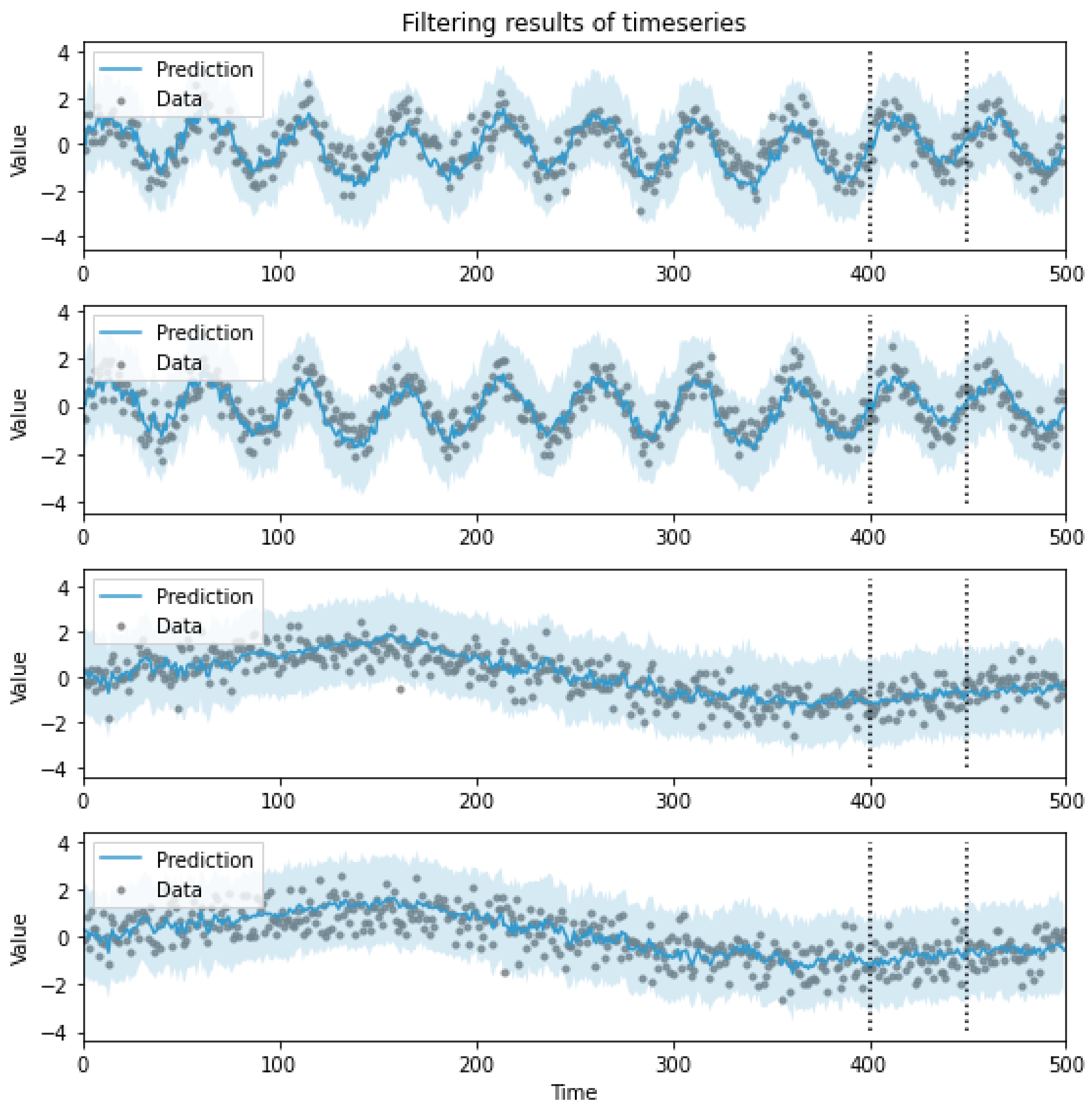
[ ]: <AxesSubplot:xlabel='Epoch'>



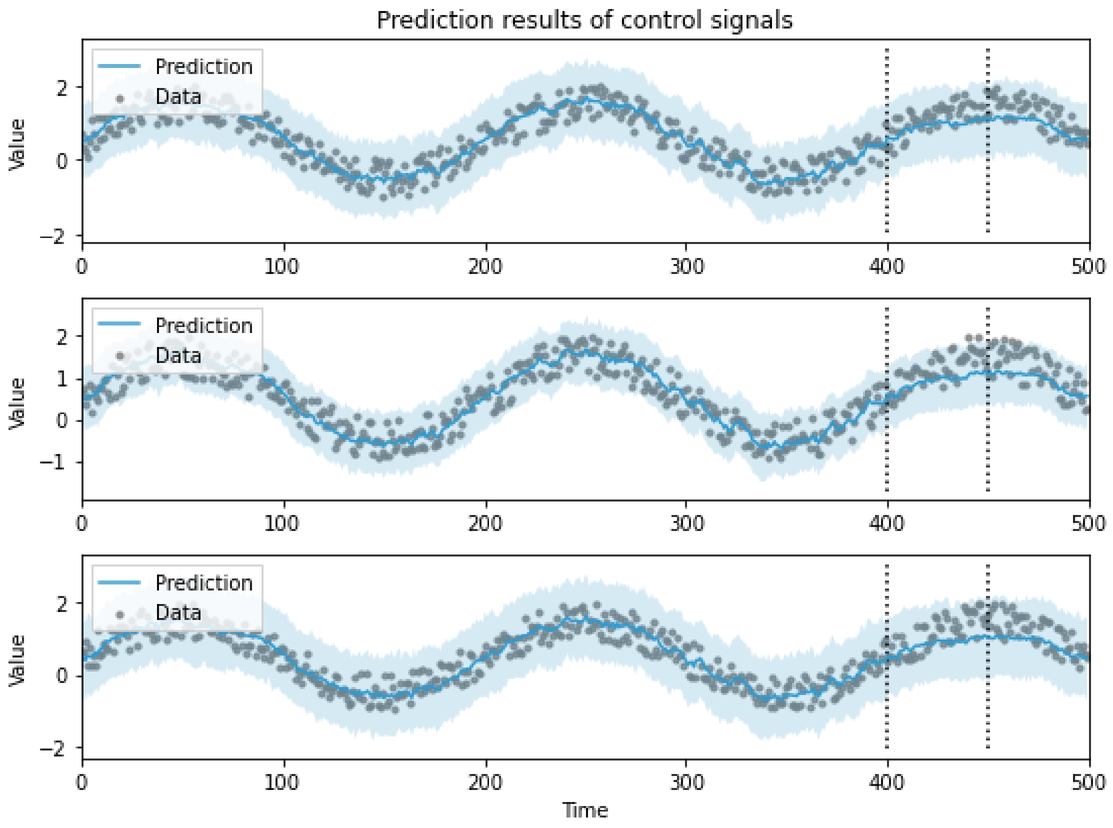
[ ]:
```
x_hat, x_025, x_975, u_hat, u_025, u_975 = dkf.predict(
    x_ten, u=u_ten, pred_steps=100, num_sample=500)

x_hat = x_hat.detach().numpy()[0]
x_025 = x_025.detach().numpy()[0]
x_975 = x_975.detach().numpy()[0]
u_hat = u_hat.detach().numpy()[0]
u_025 = u_025.detach().numpy()[0]
u_975 = u_975.detach().numpy()[0]
```

[ ]:
```
def plot_predictions(x, x_pred, x_025=None, x_975=None,
                     dims=None, val_steps=0, test_steps=0, wd=8,
                     title=None):

    assert x.ndim == 2
    if dims is None: dims = np.arange(x.shape[1])

    n_dims = len(dims)
    fig, ax = plt.subplots(n_dims, figsize=(wd, 2*n_dims))

    def plot_org(ax, xticks, x):
        ax.scatter(xticks, x, s=10, alpha=0.8, label='Data', c='#7F7F7F')
    def plot_pred(ax, xticks, x):
        ax.plot(xticks, x, c='#3399cc', label='Prediction')
    def plot_interval(ax, xticks, lower, upper):
        ax.fill_between(xticks, lower, upper, facecolor='#3399cc', alpha=0.2)
    xticks = np.arange(len(x))
    for d, axi in zip(dims, ax):

        plot_org(axi, xticks, x[:, d])
        plot_pred(axi, xticks, x_pred[:, d])
        if x_025 is not None and x_975 is not None:
            plot_interval(axi, xticks, x_025[:, d], x_975[:, d])

        ymin, ymax = axi.get_ylim()
        if val_steps > 0:
            axi.vlines(len(x) - val_steps - test_steps, ymin, ymax, ls=':',␣
↪color='black')
        if test_steps > 0:
            axi.vlines(len(x) - test_steps, ymin, ymax, ls=':', color='black')

        axi.set_xlim(0, len(x))
        axi.set_ylabel('Value')
        axi.legend(loc='upper left', fancybox=False)

    if title is not None: ax[0].set_title(title)
    ax[-1].set_xlabel('Time')
    fig.tight_layout()
```

[ ]:
```
plot_predictions(data, x_hat, x_025, x_975,
                 val_steps=50, test_steps=50,
                 title='Filtering results of timeseries')
```



```
[ ]: plot_predictions(signal, u_hat, u_025, u_975,
                     val_steps=50, test_steps=50,
                     title='Prediction results of control signals')
```


Prediction results of control signals
Prediction
Data
Value
Time
0
100
200
300
400
500
2
1
0
−1
−2