In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
%config InlineBackend.figure_formats = ['svg']
import scipy
```

In [ ]:
```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torch.utils.data import Dataset
```

L02

E 6.3

In this experiment we were asked to model the cylinder data in order to predict the following state based on the current. By doing this with machine learning we hope that the models can learn the physics that govern how the liquid flows. In order to do this I have chosen to do all the calculations (both learning and predicting) in a "r"-subspace created by the a truncated SVD. This is done since using the full 89.351 datapoints for each snapshot was too demanding for my labtop.

RNN

I will throughout this exercise be using the "UALL" attribute from the CYLINDER_ALL.mat dataset. The loading and generation of each training set (X,y) will be done with the following dataset class.

The dataset class does the following:

- Loading of the data.
- Calculates the truncated SVD and selects the first "r" columns of the U matrix to represent the subspace.
- Calculates the X values as the snapshots projected onto the subspace and scales the values according to the max value of each row.

For each model 10% of the dataset is left out to validate the result.

In [ ]:
```
class CustomDatasetRNN(Dataset):
    def __init__(self, P, test = False, r=10, num_layers = 10):
        self.num_layers = num_layers
        self.im_shape = (449,199)
```

```
        data = scipy.io.loadmat("./dmdbook_data/FLUIDS/CYLINDER_ALL.mat")

        self.u_all = torch.as_tensor(data["UALL"].astype(float))

        U, sigma, V = torch.svd(self.u_all)

        U = U[:,0:r]

        W = (torch.diag(sigma) @ V.T).T

        self.U = U[:,0:r]
        self.X = W[:,0:r]
        self.factor =  self.X.max(axis=0)[0]
        self.X = self.X / self.factor

        num_data = self.X.shape[0] - self.num_layers

        num = int(num_data * P)
        if test == False:
            self.X = self.X[:num]
        else:
            self.X = self.X[num:]

    def __len__(self):
        return self.X.shape[0] - self.num_layers

    def __getitem__(self, idx):

        X = self.X[idx : (idx + self.num_layers)]
        y = self.X[(idx + self.num_layers)]
        y_true = self.u_all[:,(idx + self.num_layers)]
        return X, y, y_true
```

In this section a RNN is implemented and trained for the dataset corospodning to predicting the next snapshot from the previous 5 states. The SVD is set to be truncated with the 20 columns corospodning to the 20 largest singular values.

In [ ]:
```
r = 20
P = 0.9
num_layers = 5

training_data = CustomDatasetRNN(P=P, r=r, num_layers=num_layers)
testing_data = CustomDatasetRNN(P=P, r=r, num_layers=num_layers, test=Tru
```

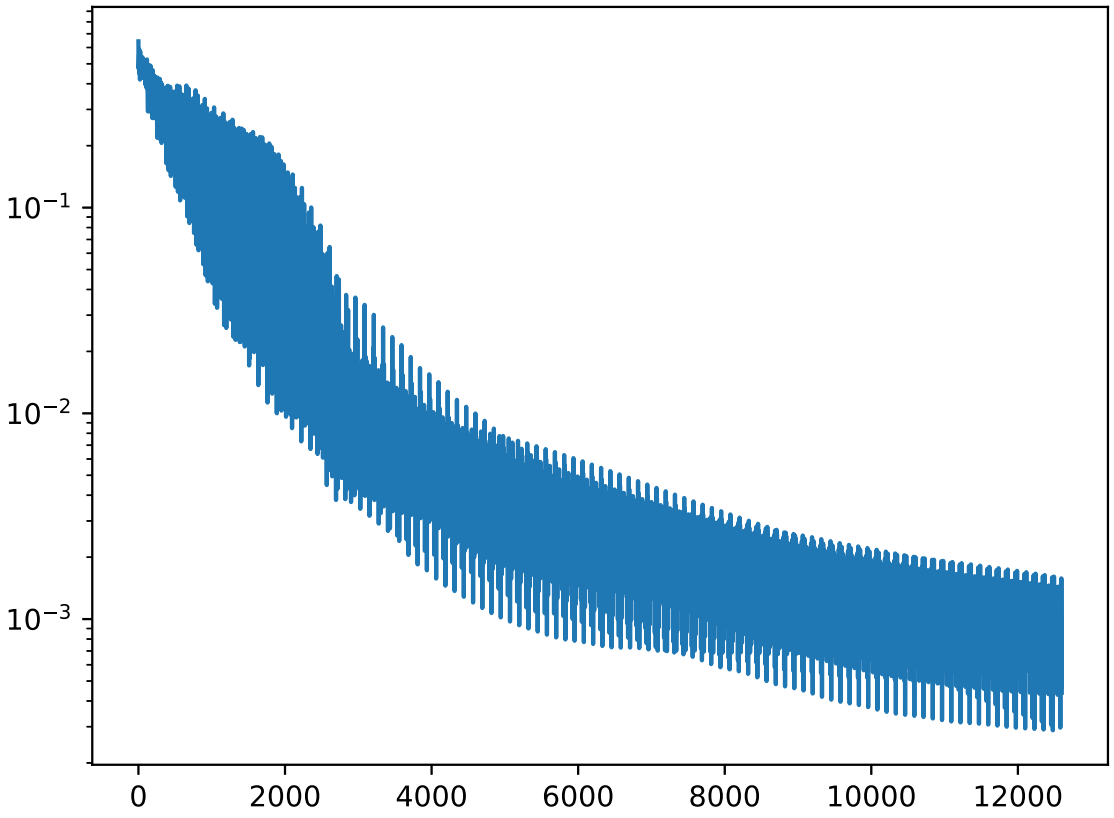
In [ ]:
```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers):
        super().__init__()
        self.r = r
        self.rnn = nn.RNN(input_size, hidden_size, num_layers).double()
        self.nn = nn.Linear(hidden_size, input_size).double()

    def forward(self, x):
```

```
        rnn_output, _ = self.rnn(x)
        nn_output = self.nn(rnn_output[-1])
        return nn_output
```

I train the dataset for 100 epochs with the stochastic gradient decent algorithm and the MSE loss.

In [ ]:
```
model = RNN(input_size=r,hidden_size=r, num_layers=num_layers).to("cpu")
loss_fn = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

train_losses = []
epochs = 100
for t in range(epochs):
    for X, y, _ in training_data:
        pred = model(X)
        loss = loss_fn(pred, y)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        train_losses.append(float(loss))

plt.semilogy(train_losses)
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x126e93390>]
```



This model can be seen to be leveling out at around a loss of 10e-3. To verify this the testing data is passed through the model. The output shows a similar avarage MSE

loss, verifying the model are capable of predicting from data it has not seen before.

(Note that MSE comparison is done in the truncated SVD subspace)

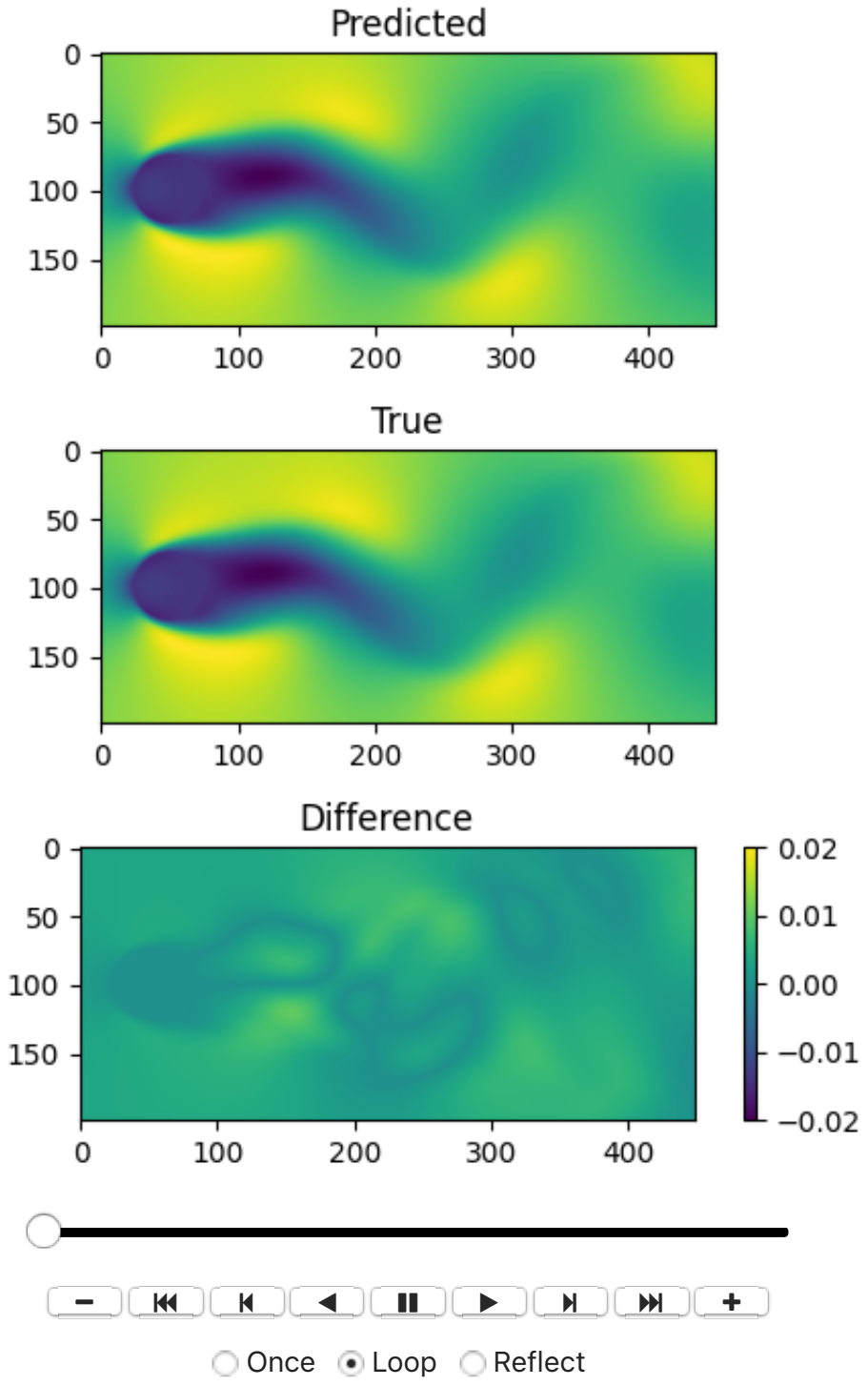
In [ ]:
```
test_losses = []

for X, y,_ in testing_data:
    pred = model(X)
    loss = loss_fn(pred, y).detach().numpy()
    test_losses.append(loss)

print(f"Average testing loss={np.mean(test_losses)}")
```

```
Average testing loss=0.001220217800170764
```

The following script produces an animation where the output of the model is transformed back into the real domain. Judging by the eyeball-norm the true and predicted look very similar.

In [ ]:
```
from matplotlib import animation
from IPython.display import HTML, display

# Reconstruct
x_reconstructed_list = []
x_true_list = []

for i in range(len(training_data)):
    X, _, y = training_data[i]

    pred = training_data.U @ (model(X) * training_data.factor)
    # y = training_data.U  @ (y * training_data.factor)
    x_reconstructed_list.append(pred.detach().numpy())
    x_true_list.append(y.detach().numpy())


# Plot
fig, axs = plt.subplots(3,1, figsize=(5,6))
fig.set_tight_layout(True)
axs[0].set_title("Predicted")
axs[1].set_title("True")
axs[2].set_title("Difference")

im1 = axs[0].imshow(x_reconstructed_list[0].reshape(training_data.im_shap
im2 = axs[1].imshow(x_true_list[0].reshape(training_data.im_shape).T)
im3 = axs[2].imshow((x_true_list[0] - x_reconstructed_list[0]).reshape(tr

plt.colorbar(im3, ax=axs[2])

def animate(i):
    im1.set_data(x_reconstructed_list[i].reshape(training_data.im_shape).
    im2.set_data(x_true_list[i].reshape(training_data.im_shape).T)
    im3.set_data(abs(x_true_list[i] - x_reconstructed_list[i]).reshape(tr
```

```
ani = animation.FuncAnimation(fig, animate, repeat=True, frames=len(train

output = HTML(ani.to_jshtml())
display(output)
plt.close()
```



The MSE for the predicted and the true values is calculated below and gives a better result as the SVD MSE.

In [ ]:
```
true_MSE = np.mean((np.array(x_true_list) - np.array(x_reconstructed_list
print(f"MSE = {true_MSE}")
```

```
MSE = 7.406714456742072e-05
```

The above animation and MSE clearly shows that using a RNN to predict snapshots is possible.

NN

The following section I implement a feedforward neural network to predict the next snapshot one based on the previous state. This is in conatrast to the RNN implementation which used the previous 5 states. The idea is that more with more parameters the network can recognize pattens that indicates how the flow behaves by only that one snapshot.

In [ ]:
```python
class CustomDatasetNN(Dataset):
    def __init__(self, P, test = False, num_inputs=10):
        self.im_shape = (449,199)

        data = scipy.io.loadmat("./dmdbook_data/FLUIDS/CYLINDER_ALL.mat")

        self.u_all = torch.as_tensor(data["UALL"].astype(float))

        U, sigma, V = torch.svd(self.u_all)

        U = U[:,0:r]

        W = (torch.diag(sigma) @ V.T).T

        self.U = U[:,0:num_inputs]
        self.X = W[:,0:num_inputs]
        self.factor =  self.X.max(axis=0)[0]
        self.X = self.X / self.factor

        self.num_data = self.X.shape[0] - 1

        num = int(self.num_data * P)
        if test == False:
            self.X = self.X[:num]
        else:
            self.X = self.X[num:]

    def __len__(self):
        return self.X.shape[0] - 1

    def __getitem__(self, idx):
        x = self.X[idx]
        y = self.X[idx+1]
        y_true = self.u_all[:, idx+1]
        return x, y, y_true
```

Like in the previous model I set the truncation number to be r=20 and the percentage

of the data used for training to be 90%.

In [ ]:
```
r = 20
P = 0.9

training_data = CustomDatasetNN(P=P, num_inputs=r)
testing_data = CustomDatasetNN(P=P, num_inputs=r, test=True)
```

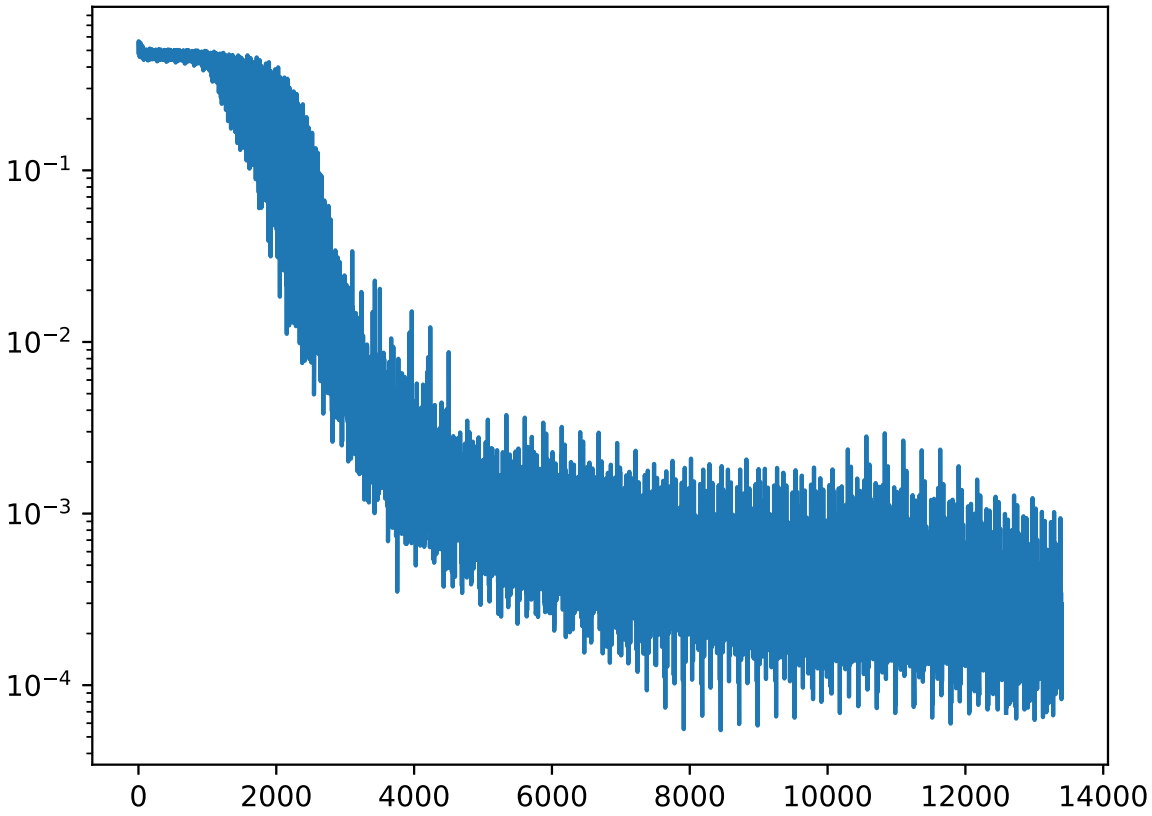
I have chosen to use a quite large network for doing this task. The network might become better/easier to optimize if this size is reduced.

In [ ]:
```
class NN(nn.Module):
    def __init__(self, num_inputs=10):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(num_inputs, 100).double(),
            nn.ReLU(),
            nn.Linear(100, 100).double(),
            nn.ReLU(),
            nn.Linear(100, 100).double(),
            nn.ReLU(),
            nn.Linear(100, 100).double(),
            nn.ReLU(),
            nn.Linear(100, 100).double(),
            nn.ReLU(),
            nn.Linear(100,num_inputs).double(),
        )

    def forward(self, x):
        logits = self.linear_relu_stack(x)
        return logits
```

Below i train the model with the SGD optimizer and the MSE loss over 100 epochs.

In [ ]:
```
model = NN(num_inputs=r).to("cpu")
loss_fn = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

train_losses = []
epochs = 100
for t in range(epochs):
    for x, y, _ in training_data:
        pred = model(x)
        loss = loss_fn(pred, y)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        train_losses.append(float(loss))

plt.semilogy(train_losses)
```

Out[ ]: [<matplotlib.lines.Line2D at 0x12abbb990>]



The plot above indicates that the FF model is better than the RNN model by having a lower MSE loss. This might be down to how long the optimization has been running for.

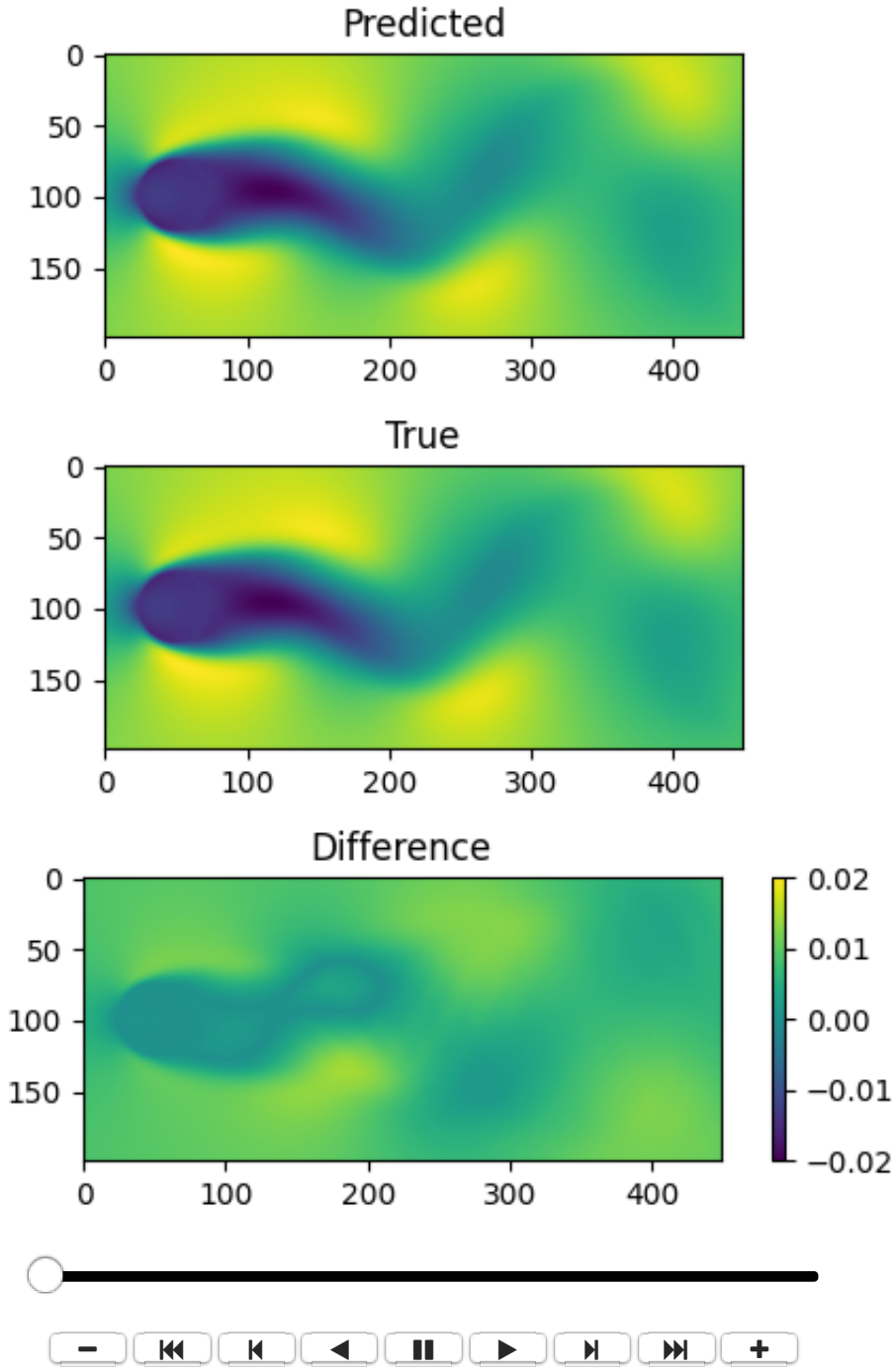
Using the test dataset the MSE loss can be verified to be around 10e-4 in the truncated SVD subspace.

In [ ]:
```
test_losses = []

for X, y, _ in testing_data:
    pred = model(X)
    loss = loss_fn(pred, y).detach().numpy()
    test_losses.append(loss)

print(f"Average testing loss={np.mean(test_losses)}")
```

```
Average testing loss=0.0004841819747622859
```

In [ ]:
```
from matplotlib import animation
from IPython.display import HTML, display

# Reconstruct
x_reconstructed_list = []
x_true_list = []

for i in range(len(training_data)):
    X, _, y = training_data[i]
```

```
    pred = training_data.U @ (model(X) * training_data.factor)
    # y = training_data.U  @ (y * training_data.factor)
    x_reconstructed_list.append(pred.detach().numpy())
    x_true_list.append(y.detach().numpy())


# Plot
fig, axs = plt.subplots(3,1, figsize=(5,6))
fig.set_tight_layout(True)
axs[0].set_title("Predicted")
axs[1].set_title("True")
axs[2].set_title("Difference")

im1 = axs[0].imshow(x_reconstructed_list[0].reshape(training_data.im_shap
im2 = axs[1].imshow(x_true_list[0].reshape(training_data.im_shape).T)
im3 = axs[2].imshow((x_true_list[0] - x_reconstructed_list[0]).reshape(tr

plt.colorbar(im3, ax=axs[2])

def animate(i):
    im1.set_data(x_reconstructed_list[i].reshape(training_data.im_shape).
    im2.set_data(x_true_list[i].reshape(training_data.im_shape).T)
    im3.set_data(abs(x_true_list[i] - x_reconstructed_list[i]).reshape(tr

ani = animation.FuncAnimation(fig, animate, repeat=True, frames=len(train

output = HTML(ani.to_jshtml())
display(output)
plt.close()
```



In [ ]:
```
true_RMSE = np.sqrt(np.mean((np.array(x_true_list) - np.array(x_reconstru
print(f"MSE = {true_MSE}")
```

```
MSE = 7.406714456742072e-05
```

The above animation and RMSE clearly shows that using a feed forward neural network to predict also is possible.

LSTM

Last I have implemented a LSTM to see if the performance of the RNN can be

improved by having a more complex model. I will again be using r=20 and 90% of the data as training and the rest as testing and validating. Like the RNN I will try to predict the snapshow based on the previous 5 states.

To get the LSTM model to converge a different optimization strategy is used. Specifically I use batching, randomized shuffle and the Adam optimizer in order to get the model to converge better.

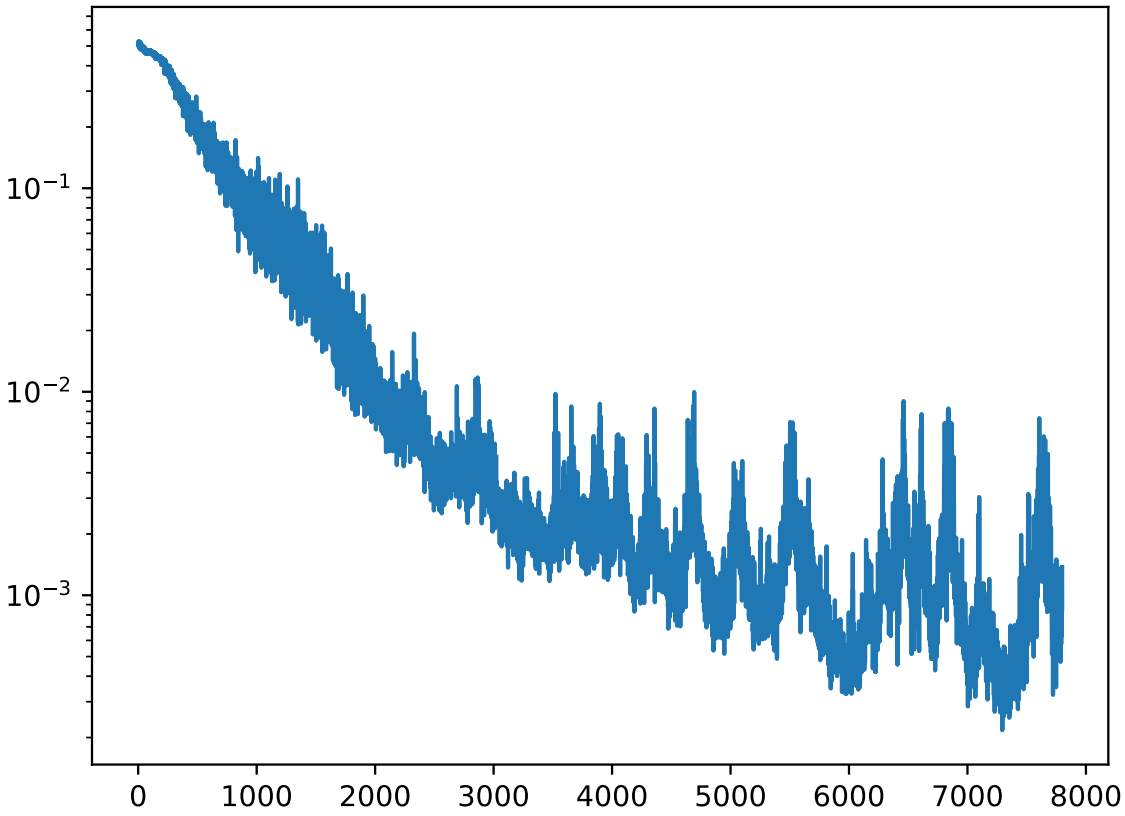
In [ ]:
```
r = 20
P = 0.9
num_layers = 5

training_data = CustomDatasetRNN(P=P, r=r, num_layers=num_layers)
training_data_loader = DataLoader(training_data, batch_size=10, shuffle=T


testing_data = CustomDatasetRNN(P=P, r=r, num_layers=num_layers, test=Tru
```

In [ ]:
```
class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers):
        super().__init__()
        self.r = r
        self.rnn = nn.LSTM(input_size, hidden_size, num_layers, batch_fir
        self.nn = nn.Linear(hidden_size, input_size).double()

    def forward(self, x):
        rnn_output, _ = self.rnn(x)
        nn_output = self.nn(rnn_output[:,-1])
        return nn_output
```

By using 600 epochs I can make the LSTM model have the same MSE loss as the RNN model.

In [ ]:
```
model = LSTM(input_size=r,hidden_size=r, num_layers=num_layers).to("cpu")
loss_fn = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)


train_losses = []
epochs = 600
for t in range(epochs):
    for X, y, _ in training_data_loader:
        pred = model(X)
        loss = loss_fn(pred, y)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        train_losses.append(loss.item())

plt.semilogy(train_losses)
```

Out[ ]: [<matplotlib.lines.Line2D at 0x12acb1e90>]



The MSE testing loss in the SVD supspace is also comparable to the RNN mse testing MSE.

In [ ]:
```
test_losses = []

for X, y, _ in testing_data:
    pred = model(X[None, :, :])
    loss = loss_fn(pred, y).detach().numpy()
    test_losses.append(loss)

print(f"Average testing loss={np.mean(test_losses)}")
```

```
Average testing loss=0.000878838849408113
```

```
/Users/louiss/code/uni/master/SML/.venv/lib/python3.11/site-packages/torch/nn/modules/loss.py:535: UserWarning: Using a target size (torch.Size([20])) that is different to the input size (torch.Size([1, 20])). This will likely lead to incorrect results due to broadcasting. Please ensure they have the same size.
  return F.mse_loss(input, target, reduction=self.reduction)
```

In [ ]:
```
x_reconstructed_list = []
x_true_list = []

for i in range(len(training_data)):
    X, _,y = training_data[i]

    pred = training_data.U @ (model(X[None, :, :]) * training_data.factor
```

```
    # y = training_data.U  @ (y * training_data.factor)
    x_reconstructed_list.append(pred.detach().numpy().flatten())
    x_true_list.append(y.detach().numpy())
```

In [ ]:
```
from matplotlib import animation
from IPython.display import HTML, display


fig, axs = plt.subplots(3,1, figsize=(5,6))
fig.set_tight_layout(True)
axs[0].set_title("Predicted")
axs[1].set_title("True")
axs[2].set_title("Difference")

im1 = axs[0].imshow(x_reconstructed_list[0].reshape(training_data.im_shap
im2 = axs[1].imshow(x_true_list[0].reshape(training_data.im_shape).T)
im3 = axs[2].imshow((x_true_list[0] - x_reconstructed_list[0]).reshape(tr
fig.colorbar(im3, ax=axs[2])

def animate(i):
    im1.set_data(x_reconstructed_list[i].reshape(training_data.im_shape).
    im2.set_data(x_true_list[i].reshape(training_data.im_shape).T)
    im3.set_data(abs(x_true_list[i] - x_reconstructed_list[i]).reshape(tr


ani = animation.FuncAnimation(fig, animate, repeat=True, frames=len(train

output = HTML(ani.to_jshtml())
display(output)
plt.close()
```