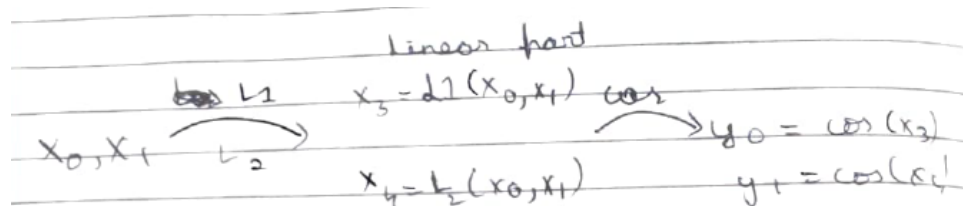


# Report-Beginners Hypothesis

## Understanding of the problem statement

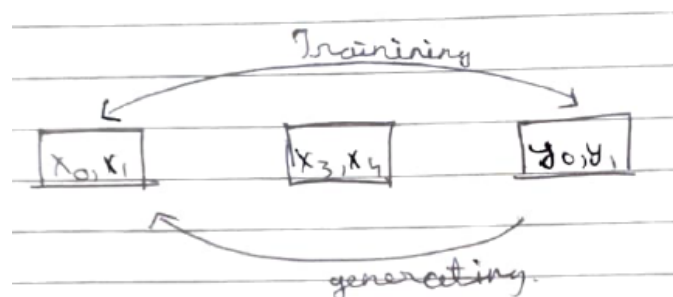
We train a CVAE where we give  $x_0$  and  $x_1$  two lines represented by 50 tensor points as the input to the encoder. The encoder encodes the information into a latent space, the dimension of it is defined by latent dimension. The decoder samples a  $z$  from the latent space and along with the conditional data ( $y_0/y_1$ ) and reconstructs  $x_0/x_1$ .

## Simplifying the problem statement



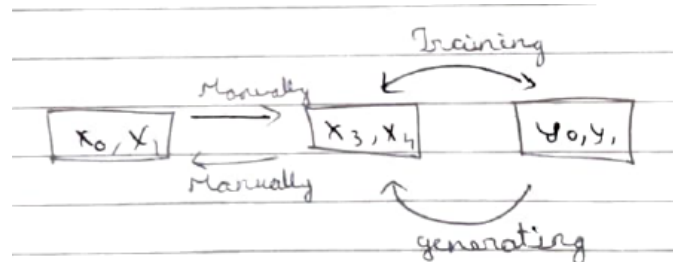
The problem statement can be considered as we first apply 2 linear functions on  $x_0$  and  $x_1$  and then we apply  $\cos$  function on both of them. We can define three approaches based on this observation.

### Approach 1:



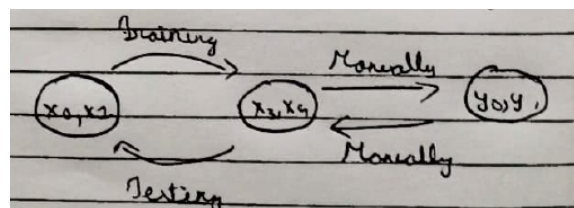
Just like the approach given in the starter notebook we train the model on  $x_0/x_1$  with  $y_0/y_1$  as the conditions. The advantage of this approach is the fact that  $x_0, x_1, y_0, y_1$  exist in the same window (i.e) between 0 and 1.

## Approach 2:



We find the  $x_3$  and  $x_4$  by applying the corresponding linear equation on  $x_0$  and  $x_1$ . We train the model with  $x_3$  and  $x_4$  as inputs with conditions as  $y_0$  and  $y_1$ . The advantage of this approach is since  $x_3$  and  $x_4$  are applied the same function  $\cos()$  the model has to learn only 1 relation between inputs and condition.

## Approach 3:



We find the  $x_3$  and  $x_4$  by applying  $\arccos$  on  $y_0$  and  $y_1$ . We train the model with  $x_0$  and  $x_1$  as inputs with conditions as  $x_3$  and  $x_4$ . The advantage of this approach is since we have removed the  $\cos$  function from the equation. The model has to now just learn the pointwise linear operation.

## Approach 2(issue with KLD)

We first changed the Signal dataset function to convert  $x_0$   $x_1$  to  $x_3$   $x_4$  across all dataloaders.

```
class SignalDataset(Dataset):
    def __init__(self, data):
        self.data = data

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        x = self.data[idx][0]
        x0 = x[:, :, 0]
        x1 = x[:, :, 1]
        x3 = 49 * x0 + 63 * x1
        x4 = 4 * x0 + 8 * x1
        x0 = x3
        x1 = x4
        x_concatenated = np.concatenate([x0[:, :, np.newaxis], x1[:, :, np.newaxis]], axis=-1)
        y = self.data[idx][1]
        return x_concatenated, y
```

Whenever we plot the lines we convert  $x_3$   $x_4$  back to  $x_0$  and  $x_1$  using inverse linear equations. However since we apply the operation on the copies of the data-loader instead of the data-loader itself the data-loader at any point of time will only contain  $x_3$   $x_4$   $y_0$   $y_1$ .

```
x,y = next(iter(dataloader))
num_samples = 30
print(x.shape)
x3 = x[:, :, :, 0]
x4 = x[:, :, :, 1]

x0 = (2/35)*x3 - (9/20)*x4
x1 = (7/20)*x4 - (1/35)*x3
```

```
x= torch.concatenate([x0[:, :, :, np.newaxis], x1[:, :, :, np.newaxis], x2[:, :, :, np.newaxis], x3[:, :, :, np.newaxis], x4[:, :, :, np.newaxis]), dim=-1)
print(x.shape,y.shape,x0.shape,x3.shape)
plot_samples(x, y, num_samples = num_samples)
```

After training the model we observe that the KDL Loss of the model is very high initially and even after training relative to the starter notebook this may be because of the different windows in which  $y_0$   $y_1$  and  $x_3 \times 4$  are present,  $y_0$  and  $y_1$  always have values between 0 and 1.  $x_3$  being  $49 \times 0 + 63 \times 1$  has values between 0 and 112.  $x_4$  being  $4 \times 0 + 8 \times 1$  has values between 0 and 12. This difference may make it harder for the model to create a meaningful latent space. We don't observe a significant increase in performance compared to the baseline.

```
Train Epoch 1: Average Loss: 21245.435573, KDL: 177435.676698, x_loss: 1749.849766, y_loss: 1.084528
{'epoch': 1, 'average_loss': 21245.435572509767, 'KLD_loss': 177435.67669757843, 'x_loss': 1749.8497659301759, 'y_loss': 1.0845279821753502}
Train Epoch 2: Average Loss: 2041.741023, KDL: 481.419311, x_loss: 995.888128, y_loss: 0.911416
{'epoch': 2, 'average_loss': 2041.7410229492189, 'KLD_loss': 481.4193107223511, 'x_loss': 995.8881280517578, 'y_loss': 0.9114157220721245}
Train Epoch 3: Average Loss: 1423.502999, KDL: 542.913727, x_loss: 683.796713, y_loss: 0.809100
{'epoch': 3, 'average_loss': 1423.5029986572265, 'KLD_loss': 542.9137268829346, 'x_loss': 683.7967126464844, 'y_loss': 0.8090995873510838}
Train Epoch 4: Average Loss: 1221.860448, KDL: 533.509027, x_loss: 583.505273, y_loss: 0.749500
{'epoch': 4, 'average_loss': 1221.860447692871, 'KLD_loss': 533.5090271759033, 'x_loss': 583.5052727508545, 'y_loss': 0.7494998374581336}
```

## Approach 3(why arc-cos sucks)

We first tried the the third approach as it looked the most promising. We apply arc cos on the tensor containing  $y$  which ensure that  $y_0$  and  $y_1$  in all data loaders are  $x_3 \times 4$ .

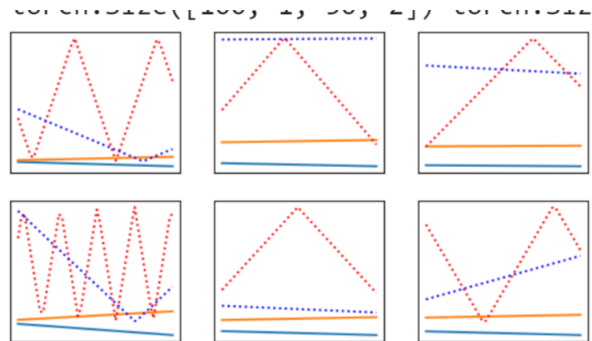
```
class SignalDataset(Dataset):
    def __init__(self, data):
        self.data = data

    def __len__(self):
        return len(self.data)
```

```
def __getitem__(self, idx):
    x = self.data[idx][0]
    y = self.data[idx][1]
    y=np.arccos(y)
    return x, y
```

X0 and X1 are a straight line with respect to index we can assume the index to be like a unit of time Since X3 and X4 are linear combinations of X0 and X1, X3 and X4 must also have a linear relation with time. However when we plot the values X3 and X4 we see that we don't get a line instead we get a piece wise line.

$$\begin{aligned} X_0 &= m_1 t + a_1 \\ X_1 &= m_2 t + a_2 \\ X_3 &= 49m_1 t + 49a_1 + 63m_2 t + 63a_2 \\ &= (49m_1 + 63m_2)t + 49a_1 + 63a_2 \\ X_4 &= (4m_1 + 8m_2)t + 4a_1 + 8a_2 \end{aligned}$$



We see that whenever the red dotted line X3 hits the boundary (0 or 1) it appears to just get reflected. This is due to the Bounded nature of arc cos. Cos is a many to one function however when we apply arc cos we only get the result which falls between -1 and 1.

To define a line we just need the slope and the intercept. We can still get the slope of the lines x3 and x4 by taking the difference of the arc cos of the first two points of the curve. However the value of the intercept can never be retrieved. This is biggest observation we found in this problem statement. Example if we try to get the intercept by taking arc cos of the first point it may give us the intercept +/- 2n(pi).

**Essentially speaking the exact value of the intercept in the  $x_3 = m_3 * t + a_3$  is lost when we apply cos over it.**

Since we get a3 from a1 and a2 the intercepts of x0 and x1. Back-tracing this we can say that the exact value(most of the information) of a1 and a2 are also lost

when apply cos . Then how can any model in general learn the intercepts ? The model can learn about the intercepts based on the reconstruction loss we define on x not from reconstruction loss on y.

**From the above logic we got the idea that improving reconstruction of x improves reconstruction of y but the converse is not true.**

The approach did not show any significant improvement over the baseline.

## Approach 1

### Loss functions:

Pearson loss: We added a loss (1- pearson coef) based on pearson coefficient to ensure that x0 and x1 are linear. But pearson loss has a max value of +1 for perfect increasing correlation and -1 for a perfect decreasing correlation. This caused both x0 and x1 to be increasing lines. We applied 1+pearson coef on x0 and 1-pearson coef on x1.

```
xr_loss = recon_loss_fn(x, recon_x)
def neg_pearson_loss(x):
    mean_x = torch.mean(x, dim=-1, keepdim=True)
    centered_x = x - mean_x
    std_x = torch.sqrt(torch.mean(centered_x ** 2, dim=-1,
    indices = torch.arange(0, x.size(-1), dtype=x.dtype, device=x.device)
    correlation = torch.mean(-1*centered_x * indices) / (std_x * std_x)
    return 1 - correlation

def pos_pearson_loss(x):
    mean_x = torch.mean(x, dim=-1, keepdim=True)
    centered_x = x - mean_x
    std_x = torch.sqrt(torch.mean(centered_x ** 2, dim=-1,
    indices = torch.arange(0, x.size(-1), dtype=x.dtype, device=x.device)
```

```

        correlation = torch.mean(centered_x * indices) / (torch.norm(centered_x))
        return 1 - correlation
x0 = recon_x[:, 0, :, 0]
x1 = recon_x[:, 0, :, 1]

pearson_loss_x0 = neg_pearson_loss(x0)
pearson_loss_x1 = pos_pearson_loss(x1)
pearson_loss=torch.mean(pearson_loss_x0+pearson_loss_x1)
x_loss=xr_loss+pearson_loss

```

cos loss: we define a addition loss for y called cos loss to ensure that y0 and y1 have cos shapes.

## Choice of hyperparameters:

Latent dim: Increase in latent dim must make the model more complex(add more non linearities to the model).We found that 100 the default value to be the suitable ,we progressively tested the model with values 25, 50,100,150,1000 and found 100 to give best performance qualitatively.

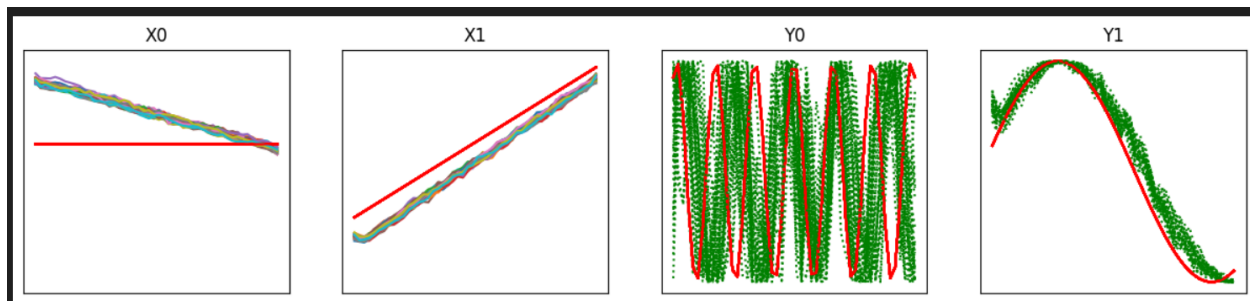
beta: we see that the loss value for KDL is already pretty low already hence the model wont improve a lot from further change in KDL. Hence we went with a value of 0.1

wx: We used the value of wx to 2 based on trial and error.

wy: We used the value of wy to 2 based on trial and error.

## A Trick(maybe not legal):

We found that after training for 20 epochs with the above settings we got a overfitting model of the solution space where the lines are almost linear 0.97. They follow the given criteria x0 is decreasing x1 is increasing and a some what better reconstruction with score 1.00 for y0 and 0.58 for y1(tier 2).But ..... the coverage was almost zero all of the 30 samples were almost overlapping



Select f0:

`cos(49 * x0 + 63 * x1)`

Select f1:

`cos(4 * x0 + 8 * x1)`

reconstruction\_error0: 1.0052459239959717

reconstruction\_error1: 0.5841844081878662

linearity\_score0: 0.9713171686253103

linearity\_score1: 0.9758833214534774

## Solution:

**Modified Rmse:** Instead of applying the regular rmse to  $x_{\text{reconstruction}}$  we can apply a modified rmse which takes into consideration deviation from multiple lines of  $x_0$  and  $x_1$  which satisfy the conditions. These values of  $x_0$  and  $x_1$  can be appeared by adding  $2 \cdot n \cdot \pi$  ( $n$  is random) to value of  $x_3$  and  $x_4$  and inverting the linear equation. However using a new loss function and changing hyperparameters according to it is time consuming instead we use a modifies `generate_sample` function.

## just add random number\*2pi:

We use the inference we learned from approach2 about the nature of the cosine inverse.



```

def generate_samples(cvae, num_samples, given_y, input_shape, zmult):

    cvae.eval()
    samples = []
    givens = []

    with torch.no_grad():
        for _ in range(num_samples):
            # Generate random latent vector
            z_rand = (torch.randn(*input_shape)*zmult)

            if torch.cuda.is_available():
                z_rand = z_rand.cuda()

            #if torch.backends.mps.is_available():
            #    z_rand = z_rand.to(torch.device('mps'))

            num_args = cvae.encoder.forward.__code__.co_argcount
            if num_args > 2 :
                z = cvae.sampling(*cvae.encoder(z_rand.unsqueeze(0)))
            else:
                z = cvae.sampling(*cvae.encoder(z_rand.unsqueeze(0)))

            # Another way to generate random latent vector
            #z = torch.randn(1, latent_dim).cuda()

            # Set conditional data as one of the given y
            # Generate sample from decoder under given_y

            sample = cvae.decoder(z, given_y)
            sample.shape
            x0 = sample[:, :, :, 0]
            x1 = sample[:, :, :, 1]
            x3 = (49 * x0) + (63 * x1)

```

```

x4 = (4 * x0) + (8 * x1)

random_number = (torch.rand(1) * 2) - 1
epsi1 = torch.full(x0.shape, random_number.item())
epsi2 = torch.full(x0.shape, random_number.item())

x3=x3+(epsi1.cuda()* 2*3.14159265358979323846264338)
x4=x4

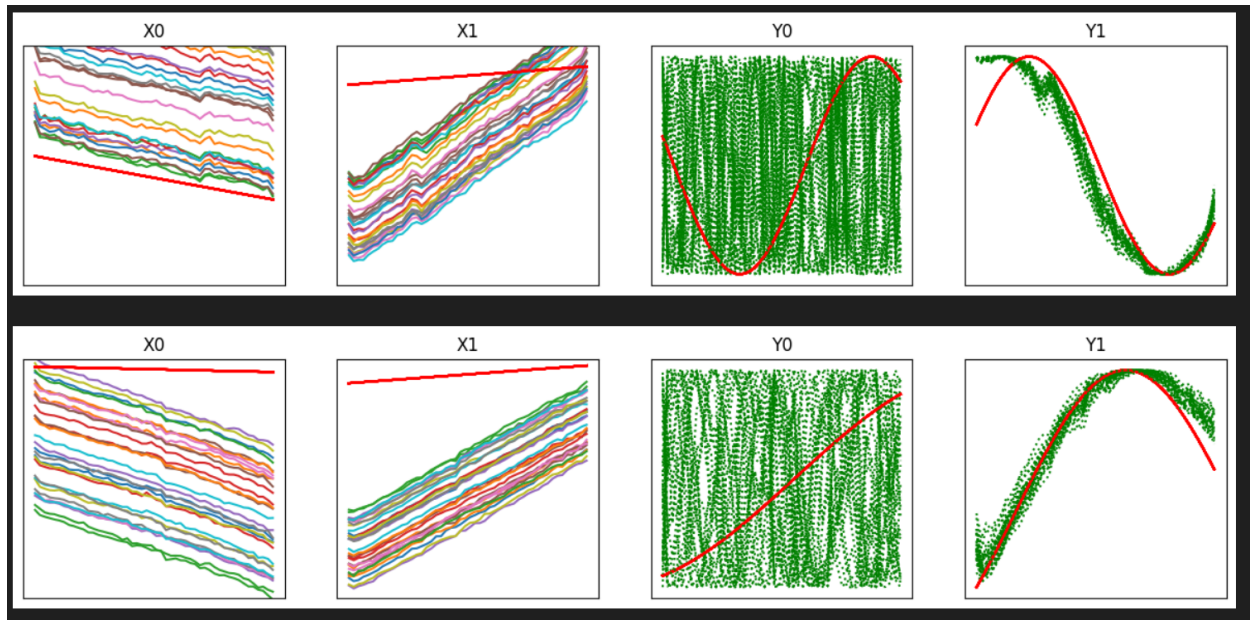
x0 =((2/35)*x3-(9/20)*x4 )
x1 =((7/20)*x4-(1/35)*x3 )
sample= torch.cat([x0[:, :, :, np.newaxis], x1[:, :, :, np.newaxis]], dim=0)

samples.append(sample)
givens.append(given_y)


samples = torch.cat(samples, dim=0)
givens = torch.cat(givens, dim=0)
return samples, givens

```

We calculate  $x_3$  and  $x_4$  from  $x_0$  and  $x_1$ . We define a random tensor  $\epsilon$  having the same shape as  $x_0$  or  $x_1$  and filled with a random number between -1.0 +1.0. We multiply  $\epsilon$  with  $2\pi$  (basically it is like  $2\pi\epsilon$  now) and we add it to  $x_3$ . We invert the linear equations and find the value of  $x_0$  and  $x_1$  again which may or not be equal to previous  $x_0$  and  $x_1$  but it will surely be parallel to it as we effectively add a constant term to each value in  $x_0$  or  $x_1$ .




We see that the quality of y1 is more or less same but the quality of y0 has decreased even further. This may be because of noise (errors of the order of  $10^{-6}$  in mathematical operations as  $\pi$  is irrational) which gets added to x0 and x1 during the above specified operations and since y0 has a higher frequency it is susceptible to the noise.



Drag and drop file here  
 Limit 200MB per file • PT

Browse files



result\_dataset (19).pt 1.1MB
 ×

Select f0:

cos(49 \* x0 + 63 \* x1)

▼

Select f1:

cos(4 \* x0 + 8 \* x1)

▼

reconstruction\_error0: 1.0170502662658691

reconstruction\_error1: 0.5820645093917847

linearity\_score0: 0.9688217125872454

linearity\_score1: 0.9692833609552176

## Approach 4(just 2 instead of 50)

We just need two things to define a line its slope and its intercept. Similarly we can say we need only the angular frequency and phase difference of a sinusoid to define it. Hence we could have a model which takes in just slopes, intercepts of  $x_0$   $x_1$  as input condition it on omega and phase difference of  $y_0$   $y_1$ . This would have vastly decreased the complexity of the data.

However this should not be possible because of the inference we get from approach 2.