# sheet06

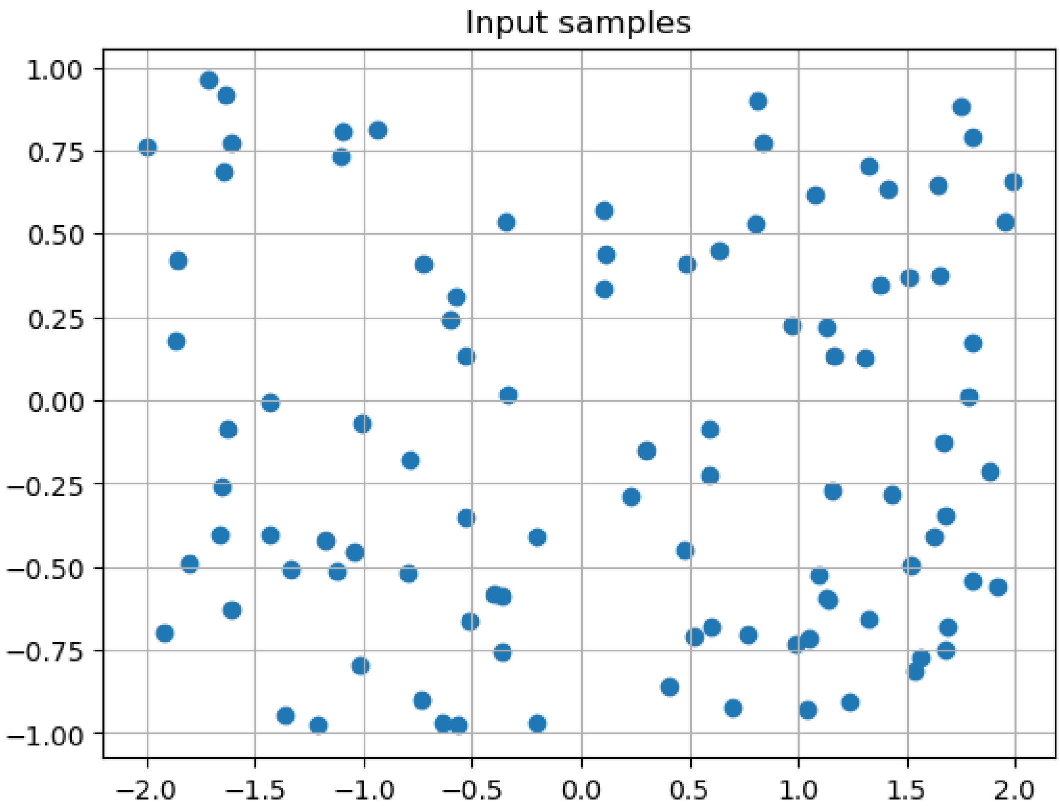
December 2, 2024

# 1 Sheet 6

## 1.1 1 Autoencoders: theory and practice

by Oliver Sange, Sam Rouppe van der Voort and Elias Huber

[7]:
```
import torch
import matplotlib.pyplot as plt
import numpy as np

# create 100 uniform samples from a rectangle [-2, 2] x [-1, 1]
num_samples = 100
data = torch.zeros(num_samples, 2)
data[:, 0] = torch.rand(num_samples) * 4 - 2
data[:, 1] = torch.rand(num_samples) * 2 - 1

# plot the samples
plt.scatter(data[:, 0], data[:, 1])
plt.title("Input samples")
plt.grid(True)
plt.show()

max_epochs_run = 1000
```




[8]:
```python
from torch.utils.data import DataLoader, TensorDataset

# Prepare data loader
dataset = TensorDataset(data, data)
data_loader = DataLoader(dataset, batch_size=8, shuffle=True, drop_last=True)

# get batched data from the data loader
x, y = next(iter(data_loader))
print("x.shape:", x.shape)
print("y.shape:", y.shape)
print("all x == y:", torch.all(x == y).item())
```

```
x.shape: torch.Size([8, 2])
y.shape: torch.Size([8, 2])
all x == y: True
```

## 1.2 (a)

[9]:
```
# TODO: define the Autoencoder architecture

import torch
from torch import nn
import pytorch_lightning as pl

class Autoencoder(nn.Module):
    def __init__(self, hidded_channels, latent_dim=1, input_dim=2,activation=nn.
 ↪ReLU,PCA=False):
        super().__init__()

        if not PCA:
            # TODO: implement the encoder and decoder
            layers_encoder = []
            layers_decoder = []
            n_layers = len(hidded_channels)

            layers_encoder.append(nn.Linear(input_dim,hidded_channels[0])) #␣
 ↪adding input layer
            #layers_encoder.append(input_dim)
            layers_encoder.append(activation())
            pass_latent = False
            for i, (l_in,l_out) in enumerate(zip(hidded_channels[:
 ↪-1],hidded_channels[1:])):
                if pass_latent:
                    layers_decoder.append(nn.Linear(l_in,l_out))
                    if i != n_layers-2: # no relu activation of output?
                        layers_decoder.append(activation())
                else:
                    if l_out == latent_dim:
                        pass_latent = True
                    layers_encoder.append(nn.Linear(l_in,l_out))
                    if not pass_latent: # no activation applied on the latent␣
 ↪space
                        layers_encoder.append(activation())

            self.encoder = nn.Sequential(*layers_encoder)
            self.decoder = nn.Sequential(*layers_decoder)
        else:
            # Linear encoder
            self.encoder = nn.Linear(input_dim, latent_dim)
            # Linear decoder
            self.decoder = nn.Linear(latent_dim, input_dim)

    def forward(self, x):
```

```
        x = self.encoder(x)
        x = self.decoder(x)
        return x

    def encode(self, x):
        x_encoded = self.encoder(x)
        return x_encoded

    def decode(self, x):
        x_decoded = self.decoder(x)
        return x_decoded

"""
            #self.encoder = nn.ModuleList(layers_encoder)
            #self.decoder = nn.ModuleList(layers_decoder)
# when using the nn.ModuleList one must define the forward pass explicitly as␣
 ↪in understood, example with loop like:
        # def forward(self, x):
        #     for layer in self.encoder:
        #         x = layer(x)
        #     for layer in self.decoder:
        #         x = layer(x)
        #     return x

"""

class AutoencoderModule(pl.LightningModule):
    def __init__(self, **model_kwargs):
        super().__init__()
        self.autoencoder = Autoencoder(**model_kwargs)
        self.loss_curve = []

    def forward(self, x):
        return self.autoencoder(x)

    def configure_optimizers(self):
        # as default use Adam optimizer:
        optimizer = torch.optim.Adam(self.parameters())#, weight_decay=1e-5)

        return optimizer

    def on_train_start(self):
        self.loss_curve = []
        return super().on_train_start()

    def training_step(self, batch):
        x, _ = batch
```

```
        x_hat = self.autoencoder(x)
        loss = nn.MSELoss()(x_hat, x)
        self.loss_curve.append(loss.item())
        return loss
```

For the Network architecture we used the ReLU activation function since we want to handle a regression problem and the ReLU is the state of the art for these classes of problems. We also decided to not put an activation fnction on the latent space since it would only half the space and project the negative encoder output to zero.

[10]:
```
hidded_channels_small = [20, 10, 1, 10, 20, 2]
autoencoder_module_small =␣
 ↪AutoencoderModule(hidded_channels=hidded_channels_small,latent_dim=1,␣
 ↪input_dim=2, activation=nn.ReLU) #Autoencoder(hidded_channels_small)#  #␣
 ↪TODO: specify the model here
print("small Model overview:", autoencoder_module_small)

hidded_channels_big = [50, 50, 50, 1, 50, 50, 50, 2]
autoencoder_module_big =␣
 ↪AutoencoderModule(hidded_channels=hidded_channels_big,latent_dim=1,␣
 ↪input_dim=2, activation=nn.ReLU )
print("big Model overview:", autoencoder_module_big)

PCA_autoencoder = AutoencoderModule(hidded_channels=None,PCA=True,latent_dim=1,␣
 ↪input_dim=2)
print("pca like Model overview:", PCA_autoencoder)
```

```
small Model overview: AutoencoderModule(
  (autoencoder): Autoencoder(
    (encoder): Sequential(
      (0): Linear(in_features=2, out_features=20, bias=True)
      (1): ReLU()
      (2): Linear(in_features=20, out_features=10, bias=True)
      (3): ReLU()
      (4): Linear(in_features=10, out_features=1, bias=True)
    )
    (decoder): Sequential(
      (0): Linear(in_features=1, out_features=10, bias=True)
      (1): ReLU()
      (2): Linear(in_features=10, out_features=20, bias=True)
      (3): ReLU()
      (4): Linear(in_features=20, out_features=2, bias=True)
    )
  )
```

```
)
big Model overview: AutoencoderModule(
  (autoencoder): Autoencoder(
    (encoder): Sequential(
      (0): Linear(in_features=2, out_features=50, bias=True)
      (1): ReLU()
      (2): Linear(in_features=50, out_features=50, bias=True)
      (3): ReLU()
      (4): Linear(in_features=50, out_features=50, bias=True)
      (5): ReLU()
      (6): Linear(in_features=50, out_features=1, bias=True)
    )
    (decoder): Sequential(
      (0): Linear(in_features=1, out_features=50, bias=True)
      (1): ReLU()
      (2): Linear(in_features=50, out_features=50, bias=True)
      (3): ReLU()
      (4): Linear(in_features=50, out_features=50, bias=True)
      (5): ReLU()
      (6): Linear(in_features=50, out_features=2, bias=True)
    )
  )
)
pca like Model overview: AutoencoderModule(
  (autoencoder): Autoencoder(
    (encoder): Linear(in_features=2, out_features=1, bias=True)
    (decoder): Linear(in_features=1, out_features=2, bias=True)
  )
)
```

### 1.3 (b)

[11]:
```
# start the training using a PyTorch Lightning Trainer
trainer_small = pl.Trainer(max_epochs=max_epochs_run,␣
↪enable_checkpointing=False) # was max_epochs=max_epochs_run (1000 for test)

trainer_small.fit(autoencoder_module_small, data_loader)
```

```
GPU available: False, used: False
TPU available: False, using: 0 TPU cores
HPU available: False, using: 0 HPUs
/home/elias/miniconda3/envs/mlph3/lib/python3.9/site-packages/pytorch_lightning/
trainer/connectors/logger_connector/logger_connector.py:75: Starting from
v1.9.0, `tensorboardX` has been removed as a dependency of the
`pytorch_lightning` package, due to potential conflicts with other packages in
the ML ecosystem. For this reason, `logger=True` will use `CSVLogger` as the
default logger, unless the `tensorboard` or `tensorboardX` packages are found.
Please `pip install lightning[extra]` or one of them to enable TensorBoard
```

```
support by default

  | Name        | Type        | Params | Mode
----------------------------------------------------
0 | autoencoder | Autoencoder | 563    | train
----------------------------------------------------
563       Trainable params
0         Non-trainable params
563       Total params
0.002     Total estimated model params size (MB)
/home/elias/miniconda3/envs/mlph3/lib/python3.9/site-
packages/pytorch_lightning/trainer/connectors/data_connector.py:424: The
'train_dataloader' does not have many workers which may be a bottleneck.
Consider increasing the value of the `num_workers` argument` to `num_workers=7`
in the `DataLoader` to improve performance.
/home/elias/miniconda3/envs/mlph3/lib/python3.9/site-
packages/pytorch_lightning/loops/fit_loop.py:298: The number of training batches
(12) is smaller than the logging interval Trainer(log_every_n_steps=50). Set a
lower value for log_every_n_steps if you want to see logs for the training
epoch.

Training: |                                                  | 0/? [00:00<?, ?it/s]

IOPub message rate exceeded.
The Jupyter server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--ServerApp.iopub_msg_rate_limit`.

Current values:
ServerApp.iopub_msg_rate_limit=1000.0 (msgs/sec)
ServerApp.rate_limit_window=3.0 (secs)

`Trainer.fit` stopped: `max_epochs=1000` reached.
```

[12]:

```python
# start the training using a PyTorch Lightning Trainer
trainer_big = pl.Trainer(max_epochs=max_epochs_run, enable_checkpointing=False)

trainer_big.fit(autoencoder_module_big, data_loader)
```

```
GPU available: False, used: False
TPU available: False, using: 0 TPU cores
HPU available: False, using: 0 HPUs

  | Name        | Type        | Params | Mode
----------------------------------------------------
0 | autoencoder | Autoencoder | 10.6 K | train
----------------------------------------------------
10.6 K    Trainable params
```

```
0         Non-trainable params
10.6 K    Total params
0.042     Total estimated model params size (MB)
```

```
Training: |                                                | 0/? [00:00<?, ?it/s]
```

```
`Trainer.fit` stopped: `max_epochs=1000` reached.
```

[13]:
```python
# start the training using a PyTorch Lightning Trainer
PCA_trainer = pl.Trainer(max_epochs=max_epochs_run, enable_checkpointing=False)

PCA_trainer.fit(PCA_autoencoder, data_loader)
```

```
GPU available: False, used: False
TPU available: False, using: 0 TPU cores
HPU available: False, using: 0 HPUs
```

```
  | Name        | Type        | Params | Mode
----------------------------------------------------
0 | autoencoder | Autoencoder | 7      | train
----------------------------------------------------
7         Trainable params
0         Non-trainable params
7         Total params
0.000     Total estimated model params size (MB)
```

```
Training: |                                                | 0/? [00:00<?, ?it/s]
```

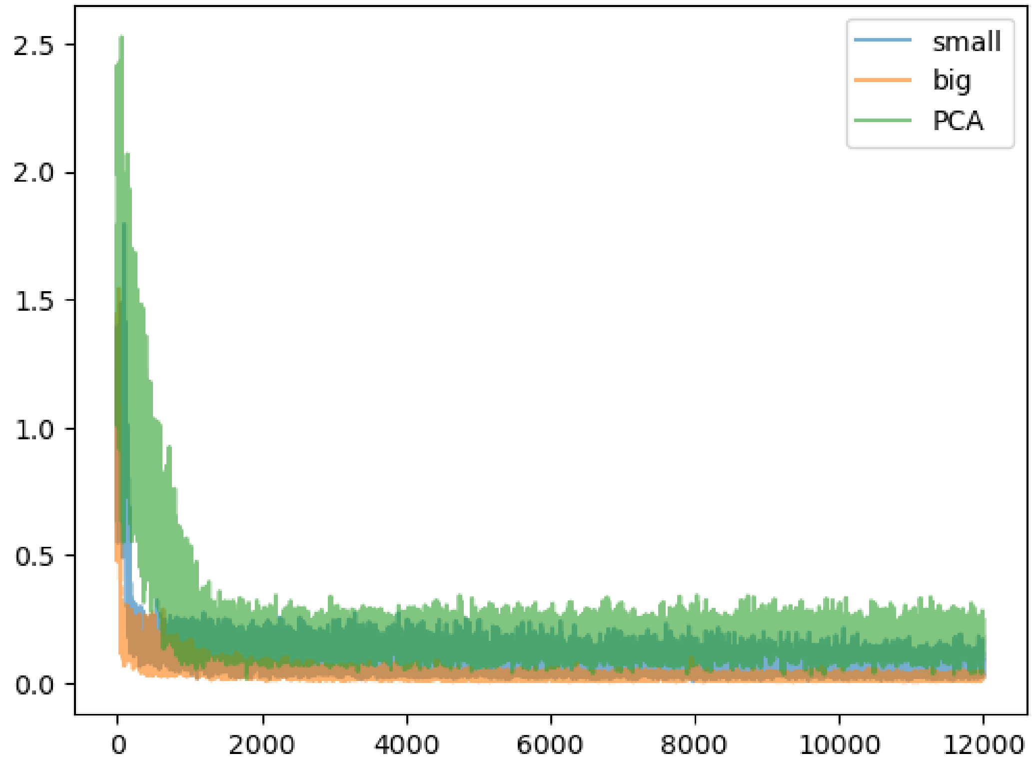
```
IOPub message rate exceeded.
The Jupyter server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--ServerApp.iopub_msg_rate_limit`.

Current values:
ServerApp.iopub_msg_rate_limit=1000.0 (msgs/sec)
ServerApp.rate_limit_window=3.0 (secs)

IOPub message rate exceeded.
The Jupyter server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--ServerApp.iopub_msg_rate_limit`.

Current values:
ServerApp.iopub_msg_rate_limit=1000.0 (msgs/sec)
ServerApp.rate_limit_window=3.0 (secs)

IOPub message rate exceeded.
The Jupyter server will temporarily stop sending output
```

```
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--ServerApp.iopub_msg_rate_limit`.

Current values:
ServerApp.iopub_msg_rate_limit=1000.0 (msgs/sec)
ServerApp.rate_limit_window=3.0 (secs)

IOPub message rate exceeded.
The Jupyter server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--ServerApp.iopub_msg_rate_limit`.

Current values:
ServerApp.iopub_msg_rate_limit=1000.0 (msgs/sec)
ServerApp.rate_limit_window=3.0 (secs)

IOPub message rate exceeded.
The Jupyter server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--ServerApp.iopub_msg_rate_limit`.

Current values:
ServerApp.iopub_msg_rate_limit=1000.0 (msgs/sec)
ServerApp.rate_limit_window=3.0 (secs)

IOPub message rate exceeded.
The Jupyter server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--ServerApp.iopub_msg_rate_limit`.

Current values:
ServerApp.iopub_msg_rate_limit=1000.0 (msgs/sec)
ServerApp.rate_limit_window=3.0 (secs)

IOPub message rate exceeded.
The Jupyter server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--ServerApp.iopub_msg_rate_limit`.

Current values:
ServerApp.iopub_msg_rate_limit=1000.0 (msgs/sec)
ServerApp.rate_limit_window=3.0 (secs)
```

```
IOPub message rate exceeded.
The Jupyter server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--ServerApp.iopub_msg_rate_limit`.

Current values:
ServerApp.iopub_msg_rate_limit=1000.0 (msgs/sec)
ServerApp.rate_limit_window=3.0 (secs)

IOPub message rate exceeded.
The Jupyter server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--ServerApp.iopub_msg_rate_limit`.

Current values:
ServerApp.iopub_msg_rate_limit=1000.0 (msgs/sec)
ServerApp.rate_limit_window=3.0 (secs)

`Trainer.fit` stopped: `max_epochs=1000` reached.
```

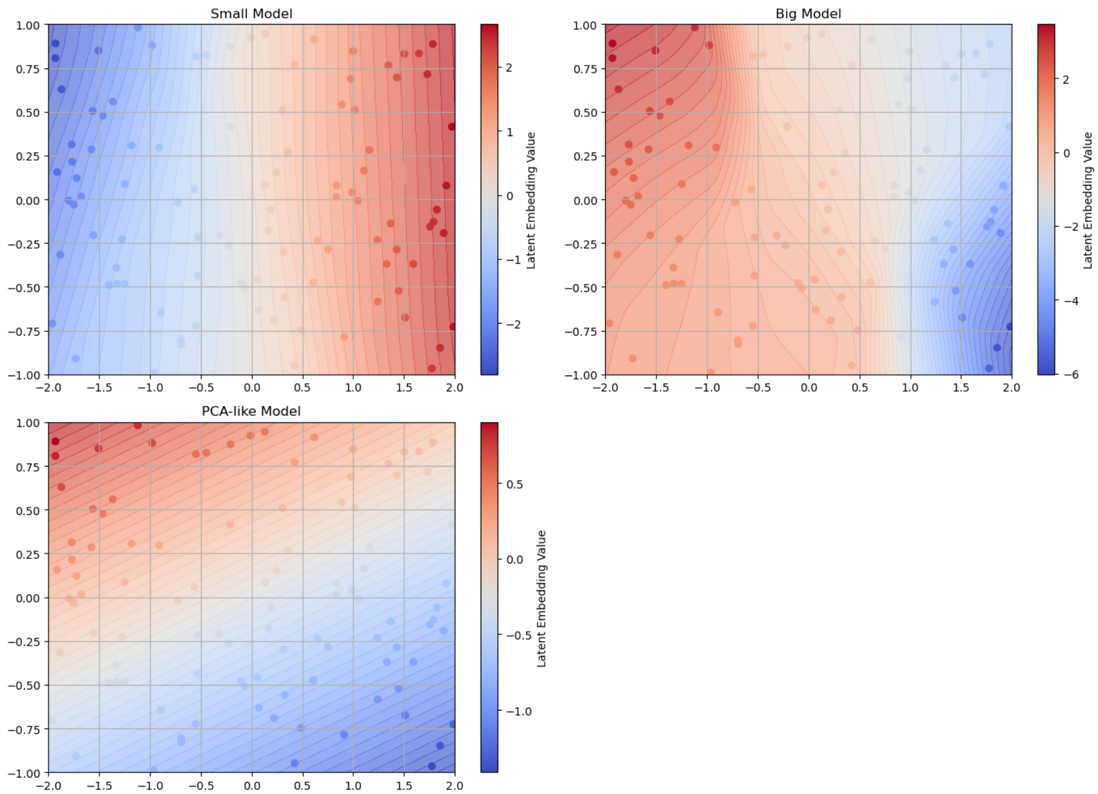
[14]:
```
# len(autoencoder_module_big.loss_curve)
## here im not completely sure what i should actually plot, probably average␣
↪over each epoch makes more sense,
# i dont understand why the loss_curve length is so long.
plt.plot(range(len(autoencoder_module_small.
↪loss_curve)),autoencoder_module_small.loss_curve,label="small",alpha = 0.6)
plt.plot(range(len(autoencoder_module_big.loss_curve)),autoencoder_module_big.
↪loss_curve,label="big",alpha = 0.6)
plt.plot(range(len(PCA_autoencoder.loss_curve)),PCA_autoencoder.
↪loss_curve,label="PCA",alpha = 0.6)
plt.legend()
```

[14]:
```
<matplotlib.legend.Legend at 0x7fda4c5076a0>
```




[40]:
```
latent_embedding_small = autoencoder_module_small.autoencoder.encode(data)
latent_embedding_big = autoencoder_module_big.autoencoder.encode(data)
latent_embedding_PCA = PCA_autoencoder.autoencoder.encode(data)

# Flatten the grid for encoding
x_grid, y_grid = np.meshgrid(np.linspace(-2, 2, 100), np.linspace(-1, 1, 100))
# Convert grid to PyTorch tensors
x_tensor = torch.from_numpy(x_grid).float()  # Shape: (100, 100)
y_tensor = torch.from_numpy(y_grid).float()  # Shape: (100, 100)

# Step 2: Combine x and y into a grid tensor for encoding
grid = torch.stack((x_tensor, y_tensor), dim=-1)

grid_flat = grid.view(-1, 2)  # Flatten to shape (10000, 2)

# Encode the grid points using the autoencoders
contour_values1 = autoencoder_module_small.autoencoder.encode(grid_flat)
contour_values2 = autoencoder_module_big.autoencoder.encode(grid_flat)
contour_values3 = PCA_autoencoder.autoencoder.encode(grid_flat)

# Reshape the encoded values back to a 2D grid for contourf
contour_values1 = contour_values1.view(100, 100).detach().numpy()
```

```
contour_values2 = contour_values2.view(100, 100).detach().numpy()
contour_values3 = contour_values3.view(100, 100).detach().numpy()

# Plot
fig, axes = plt.subplots(2, 2, figsize=(14, 10))  # Adjust figsize for better layout

# Plot latent embeddings for each model
contour1 = axes[0, 0].contourf(x_grid, y_grid, contour_values1, levels=50, cmap='coolwarm', alpha=0.7)
contour2 = axes[0, 1].contourf(x_grid, y_grid, contour_values2, levels=50, cmap='coolwarm', alpha=0.7)
contour3 = axes[1, 0].contourf(x_grid, y_grid, contour_values3, levels=50, cmap='coolwarm', alpha=0.7)

scatter1 = axes[0, 0].scatter(data[:, 0], data[:, 1], c=latent_embedding_small.detach().numpy(), cmap='coolwarm')
scatter2 = axes[0, 1].scatter(data[:, 0], data[:, 1], c=latent_embedding_big.detach().numpy(), cmap='coolwarm')
scatter3 = axes[1, 0].scatter(data[:, 0], data[:, 1], c=latent_embedding_PCA.detach().numpy(), cmap='coolwarm')

fig.colorbar(scatter1, ax=axes[0, 0], label="Latent Embedding Value")
fig.colorbar(scatter2, ax=axes[0, 1], label="Latent Embedding Value")
fig.colorbar(scatter3, ax=axes[1, 0], label="Latent Embedding Value")

axes[0, 0].set_title("Small Model")
axes[0, 1].set_title("Big Model")
axes[1, 0].set_title("PCA-like Model")
axes[0, 0].grid(True)
axes[0, 1].grid(True)
axes[1, 0].grid(True)

# Hide the empty subplot (bottom-right) as we only need 3 plots
axes[1, 1].axis('off')
plt.tight_layout()
plt.show()
```




The PCA model prefers a color gradient around the X2 axis, the X1 axis seems to influence the embedding less. This can be due to getting the X2 axis wrong leads to a higher loss than for the X1 axis, due to the stretch in the sample space.

## 1.4 (c)

## 1.5 also make a prediciton before looking at plots below

If we sample points from an interval in the latent space, the decoder can attempt to map them to a curve in the ambient/input space. Guess what these curves may look like: i) After random initialization of the MLP parameters, and ii) After training the respective architecture.

### 1.5.1 i)

A random walk like line.

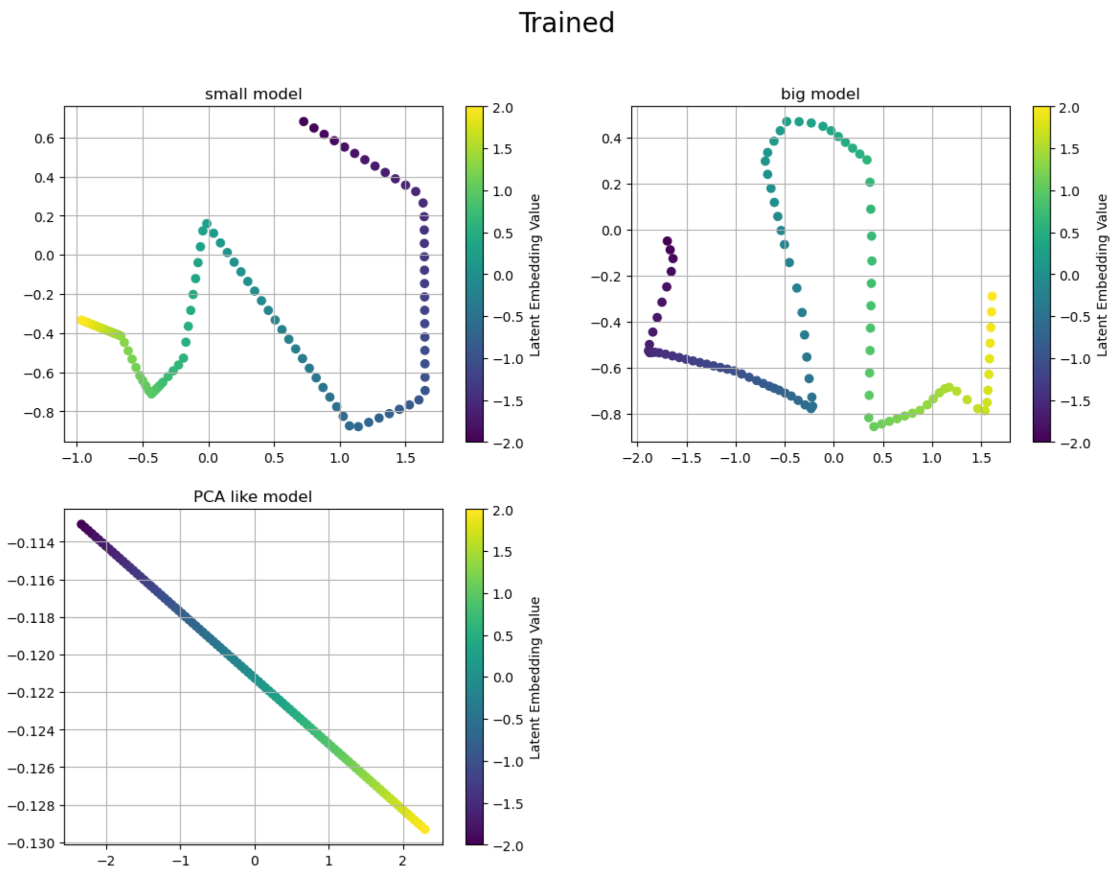
### 1.5.2 ii)

We think its a line following the color change seen above.

### 1.6 (d)

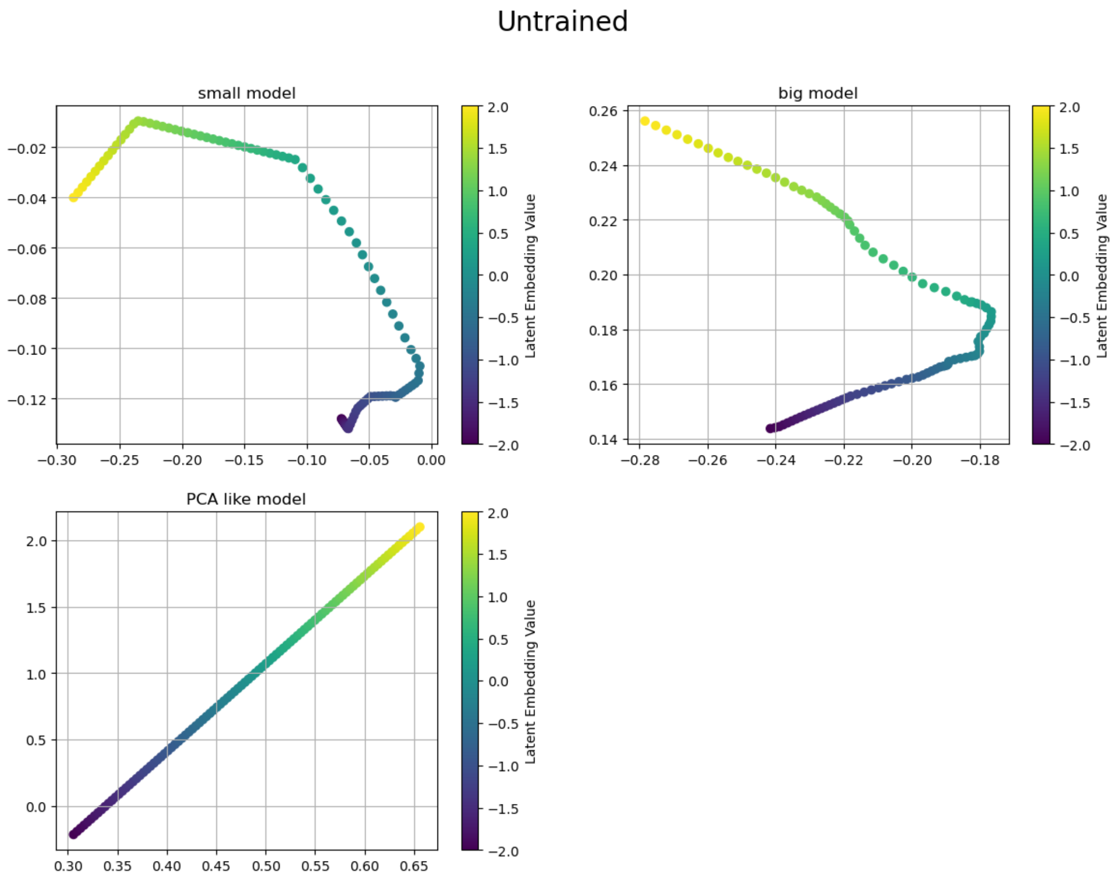
[15]:
```
import numpy as np

latent_space_samples = np.linspace(-2, 2, 100) # somewhat arbitrary choice for␣
↪ReLu maybe, shoud just inlude origin
# if we change the non linearity to sigmoid, then from -1 to 1 i would say,
# but also we need to think if we should have a non linearity in latent space
# for this random num task i dont see why not but idk
latent_space_samples = torch.tensor(latent_space_samples,dtype=torch.float32).
↪view(100, 1)


samples_decoded_trained_small = autoencoder_module_small.autoencoder.
↪decode(latent_space_samples)
samples_decoded_trained_big = autoencoder_module_big.autoencoder.
↪decode(latent_space_samples)
samples_decoded_trained_PCA = PCA_autoencoder.autoencoder.
↪decode(latent_space_samples)

fig, axes = plt.subplots(2, 2, figsize=(14, 10))  # Adjust figsize for better␣
↪layout

# Plot latent embeddings for each model

scatter1 = axes[0,0].scatter(samples_decoded_trained_small[:, 0].detach().
↪numpy(), samples_decoded_trained_small[:, 1].detach().
↪numpy(),c=latent_space_samples.detach().numpy(), cmap='viridis')
scatter2 = axes[0,1].scatter(samples_decoded_trained_big[:, 0].detach().
↪numpy(), samples_decoded_trained_big[:, 1].detach().
↪numpy(),c=latent_space_samples.detach().numpy(), cmap='viridis')
scatter3 = axes[1,0].scatter(samples_decoded_trained_PCA[:, 0].detach().
↪numpy(), samples_decoded_trained_PCA[:, 1].detach().
↪numpy(),c=latent_space_samples.detach().numpy(), cmap='viridis')

fig.colorbar(scatter1, ax=axes[0, 0], label="Latent Embedding Value")
fig.colorbar(scatter2, ax=axes[0, 1], label="Latent Embedding Value")
fig.colorbar(scatter3, ax=axes[1, 0], label="Latent Embedding Value")

axes[0,0].set_title("small model")
axes[0,1].set_title("big model")
axes[1,0].set_title("PCA like model")

axes[0,0].grid(True)
axes[0,1].grid(True)
axes[1,0].grid(True)
```

```
fig.suptitle('Trained',fontsize=20)
# Hide the empty subplot (bottom-right) as we only need 3 plots
axes[1, 1].axis('off')
plt.show()
```




[16]:
```
# init untrained models
untrained_autoencoder_small = AutoencoderModule(hidded_channels=hidded_channels_small,latent_dim=1, input_dim=2, activation=nn.ReLU)
untrained_autoencoder_big = AutoencoderModule(hidded_channels=hidded_channels_big,latent_dim=1, input_dim=2, activation=nn.ReLU )
untrained_PCA_autoencoder = AutoencoderModule(hidded_channels=None,PCA=True,latent_dim=1, input_dim=2)

samples_decoded_untrained_small = untrained_autoencoder_small.autoencoder.decode(latent_space_samples)
samples_decoded_untrained_big = untrained_autoencoder_big.autoencoder.decode(latent_space_samples)
```

```
samples_decoded_untrained_PCA = untrained_PCA_autoencoder.autoencoder.
 ↪decode(latent_space_samples)

fig, axes = plt.subplots(2, 2, figsize=(14, 10))  # Adjust figsize for better␣
 ↪layout


scatter1 = axes[0,0].scatter(samples_decoded_untrained_small[:, 0].detach().
 ↪numpy(), samples_decoded_untrained_small[:, 1].detach().
 ↪numpy(),c=latent_space_samples.detach().numpy(), cmap='viridis')
scatter2 = axes[0,1].scatter(samples_decoded_untrained_big[:, 0].detach().
 ↪numpy(), samples_decoded_untrained_big[:, 1].detach().
 ↪numpy(),c=latent_space_samples.detach().numpy(), cmap='viridis')
scatter3 = axes[1,0].scatter(samples_decoded_untrained_PCA[:, 0].detach().
 ↪numpy(), samples_decoded_untrained_PCA[:, 1].detach().
 ↪numpy(),c=latent_space_samples.detach().numpy(), cmap='viridis')

fig.colorbar(scatter1, ax=axes[0, 0], label="Latent Embedding Value")
fig.colorbar(scatter2, ax=axes[0, 1], label="Latent Embedding Value")
fig.colorbar(scatter3, ax=axes[1, 0], label="Latent Embedding Value")

axes[0,0].set_title("small model")
axes[0,1].set_title("big model")
axes[1,0].set_title("PCA like model")

axes[0,0].grid(True)
axes[0,1].grid(True)
axes[1,0].grid(True)
fig.suptitle('Untrained',fontsize=20)
# Hide the empty subplot (bottom-right) as we only need 3 plots
axes[1, 1].axis('off')
plt.show()
```




We observe that the trained model curve tries to cover most of the space where the data points are placed. This is contrary to the untrained model where the region where the path goes is stretched on a smaller region.

## 1.7 (e) and (f) discussion

### 1.7.1 (e)

Given enough parameters, the encoder can reconstruct the finite number of points by fitting the intepolation polynimal that fits all the points. This representation however is not very useful due to extreme overfitting.

An MLP autoencoder with a bottleneck dimension of 1 can theoretically reconstruct all data points given a sufficiently complex architecture and effective training. Since MLPs are universal approximators, they can approximate the necessary mappings if they have enough width and depth. However, practical challenges, such as optimization difficulties or finite numerical precision, can affect the model's performance. While perfect reconstruction theoretically possible, achieving it in practice is often very difficult.

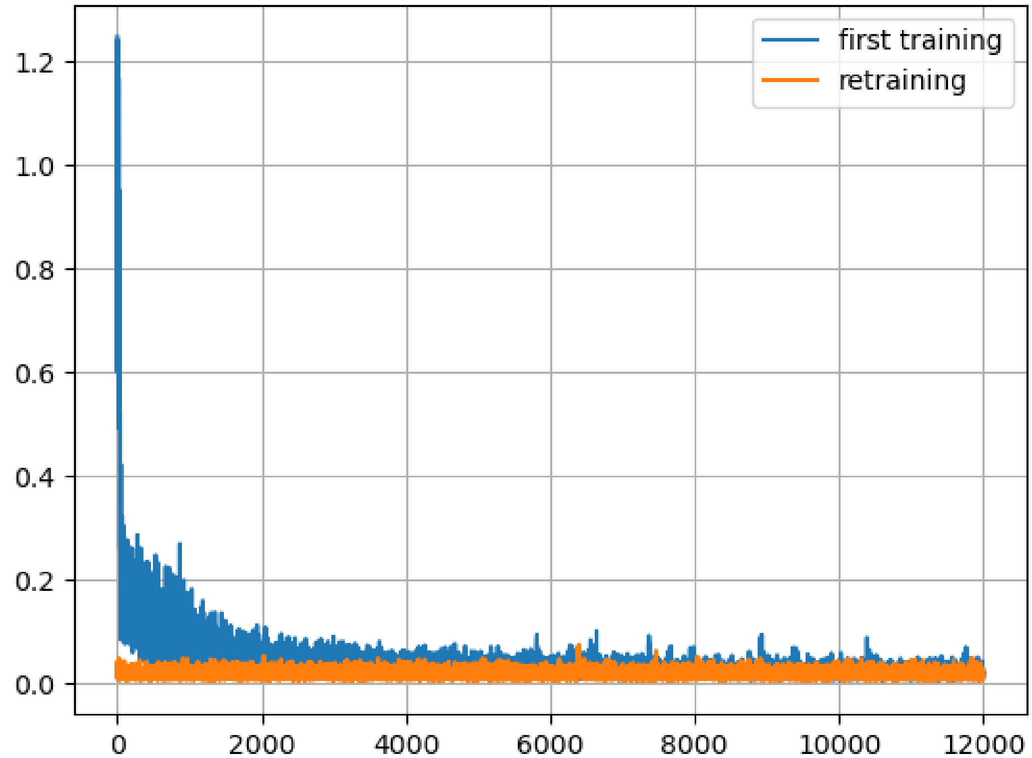
### 1.7.2 (f)

The encoder will with high likelyhood converge to a embedding close to the original encoder. But due to inefficiencies during training and the now less possible output values (since the decoder is fixed) the result will be less accurate then the original encoder.

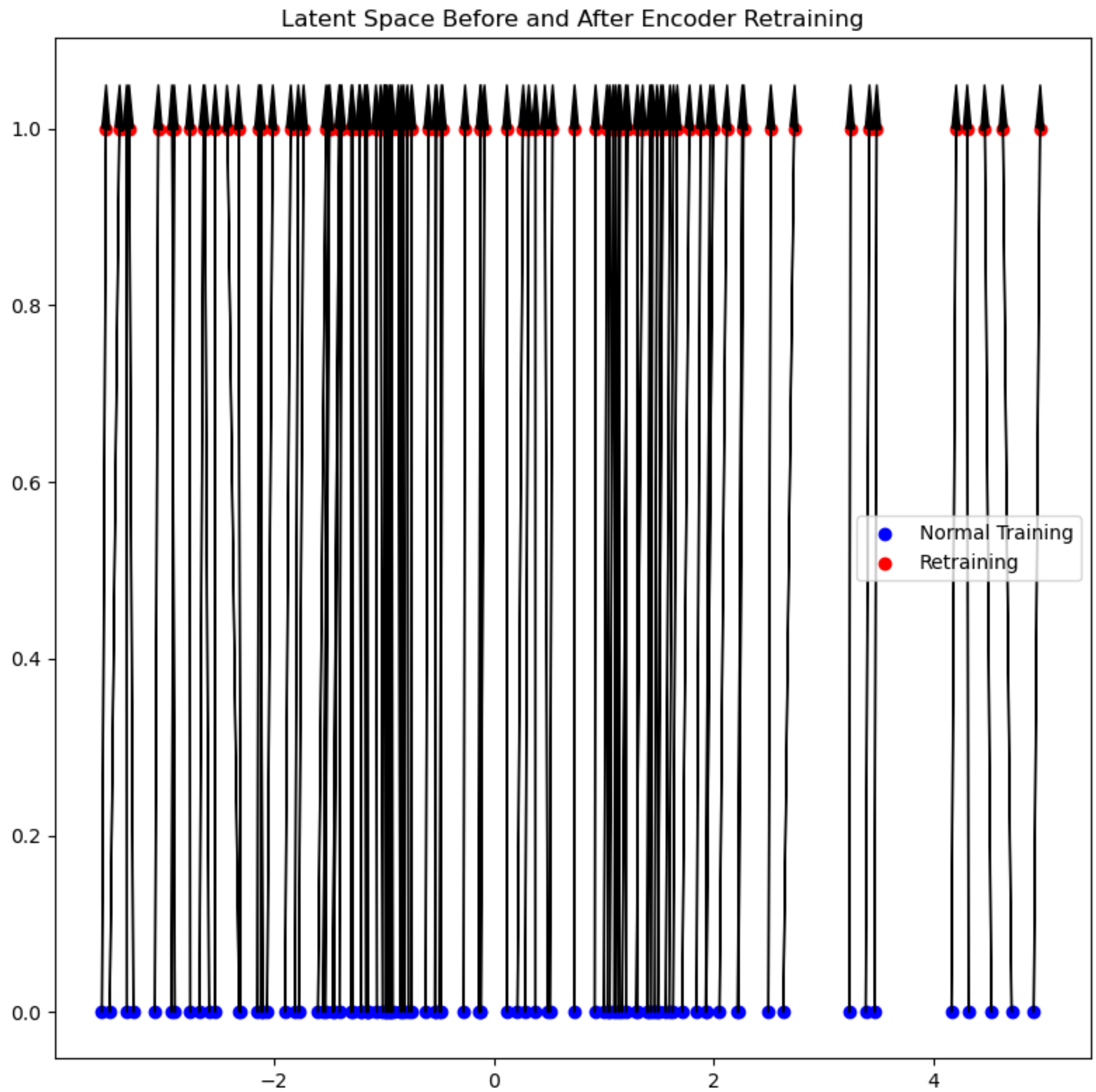
[17]:
```
hidded_channels_big = [50, 50, 50, 1, 50, 50, 50, 2]
autoencoder_module_big_retrain = AutoencoderModule(hidded_channels=hidded_channels_big,latent_dim=1, input_dim=2, activation=nn.ReLU )

# start the training using a PyTorch Lightning Trainer
retrain_trainer = pl.Trainer(max_epochs=max_epochs_run, enable_checkpointing=False)

retrain_trainer.fit(autoencoder_module_big_retrain, data_loader)
plt.plot(autoencoder_module_big_retrain.loss_curve,label="first training")

latent_space_data_trained = autoencoder_module_big_retrain.autoencoder.encoder(data).detach().numpy()

#now fix decoder parameters
for parameter in autoencoder_module_big_retrain.autoencoder.decoder.parameters():
    parameter.requires_grad = False

# reinitialize parameters for the encoder
for parameter in autoencoder_module_big_retrain.autoencoder.encoder.parameters():
    if isinstance(parameter,nn.Linear):
        nn.init.normal_(parameter.weight,mean=0.0,std=1)
        nn.init.zeros_(parameter.bias)

retrain_trainer = pl.Trainer(max_epochs=max_epochs_run, enable_checkpointing=False)
retrain_trainer.fit(autoencoder_module_big_retrain, data_loader)
plt.plot(autoencoder_module_big_retrain.loss_curve,label="retraining")
plt.legend()
plt.grid()
plt.show()

latent_space_data_retrained = autoencoder_module_big_retrain.autoencoder.encoder(data).detach().numpy()
```

```
GPU available: False, used: False
TPU available: False, using: 0 TPU cores
HPU available: False, using: 0 HPUs
```

```
  | Name        | Type        | Params | Mode
---------------------------------------------------
0 | autoencoder | Autoencoder | 10.6 K | train
---------------------------------------------------
10.6 K    Trainable params
0         Non-trainable params
10.6 K    Total params
0.042     Total estimated model params size (MB)

Training: |                                                   | 0/? [00:00<?, ?it/s]

`Trainer.fit` stopped: `max_epochs=1000` reached.
GPU available: False, used: False
TPU available: False, using: 0 TPU cores
HPU available: False, using: 0 HPUs

  | Name        | Type        | Params | Mode
---------------------------------------------------
0 | autoencoder | Autoencoder | 10.6 K | train
---------------------------------------------------
5.3 K     Trainable params
5.3 K     Non-trainable params
10.6 K    Total params
0.042     Total estimated model params size (MB)

Training: |                                                   | 0/? [00:00<?, ?it/s]

IOPub message rate exceeded.
The Jupyter server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--ServerApp.iopub_msg_rate_limit`.

Current values:
ServerApp.iopub_msg_rate_limit=1000.0 (msgs/sec)
ServerApp.rate_limit_window=3.0 (secs)

IOPub message rate exceeded.
The Jupyter server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--ServerApp.iopub_msg_rate_limit`.

Current values:
ServerApp.iopub_msg_rate_limit=1000.0 (msgs/sec)
ServerApp.rate_limit_window=3.0 (secs)

`Trainer.fit` stopped: `max_epochs=1000` reached.
```




[21]:
```
fig, ax = plt.subplots(figsize=(8, 8))
ax.scatter(latent_space_data_trained[:, 0], np.
 ↪zeros_like(latent_space_data_trained[:, 0]), color='blue',
           label='Normal Training')
ax.scatter(latent_space_data_retrained[:, 0], np.
 ↪zeros_like(latent_space_data_retrained[:, 0])+1,
           color='red', label='Retraining')
# Add arrows showing how the encoder's output changed after retraining
for i in range(len(latent_space_data_trained)):
    ax.arrow(latent_space_data_trained[i, 0], 0, latent_space_data_retrained[i,␣
 ↪0] -
             latent_space_data_trained[i,0], 1, head_width=0.07, head_length=0.
 ↪05, fc='black', ec='black')
ax.set_title('Latent Space Before and After Encoder Retraining')
ax.legend()
plt.tight_layout()
plt.show()
```




## 1.8 (h)

[22]:
```python
# define model that uses SGD
class AutoencoderModuleSGD(pl.LightningModule):
    def __init__(self, **model_kwargs):
        super().__init__()
        self.autoencoder = Autoencoder(**model_kwargs)
        self.loss_curve = []

    def forward(self, x):
        return self.autoencoder(x)
```

```
    def configure_optimizers(self):
        # as default use Adam optimizer:
        optimizer = torch.optim.SGD(self.parameters(), weight_decay=1e-5)

        return optimizer

    def on_train_start(self):
        self.loss_curve = []
        return super().on_train_start()

    def training_step(self, batch):
        x, _ = batch
        x_hat = self.autoencoder(x)
        loss = nn.MSELoss()(x_hat, x)
        self.loss_curve.append(loss.item())
        return loss
```

[26]:

```
hidded_channels_big = [50, 50, 50, 1, 50, 50, 50, 2]
autoencoder_module_big_retrain =␣
 ↪AutoencoderModule(hidded_channels=hidded_channels_big,latent_dim=1,␣
 ↪input_dim=2, activation=nn.ReLU)
dataset = TensorDataset(data, data)
data_loader = DataLoader(dataset, batch_size=100, shuffle=True, drop_last=True)

num_training_examples = [list(range(8, 8*12*max_epochs_run+1, 8)),␣
 ↪list(range(100,100*1*max_epochs_run+1, 100))]
#num training samples given by step=num batch size, and total␣
 ↪train_data_seen=batch_size*num_batches*epochs

# start the training using a PyTorch Lightning Trainer
retrain_trainer = pl.Trainer(max_epochs=max_epochs_run,␣
 ↪enable_checkpointing=False)

retrain_trainer.fit(autoencoder_module_big_retrain, data_loader)
plt.plot(num_training_examples[1],autoencoder_module_big_retrain.
 ↪loss_curve,label="first training GD",alpha = 0.6)

#now fix decoder parameters
for parameter in autoencoder_module_big_retrain.autoencoder.decoder.
 ↪parameters():
    parameter.requires_grad = False

# reinitialize parameters for the encoder
for parameter in autoencoder_module_big_retrain.autoencoder.encoder.
 ↪parameters():
    if isinstance(parameter,nn.Linear):
```

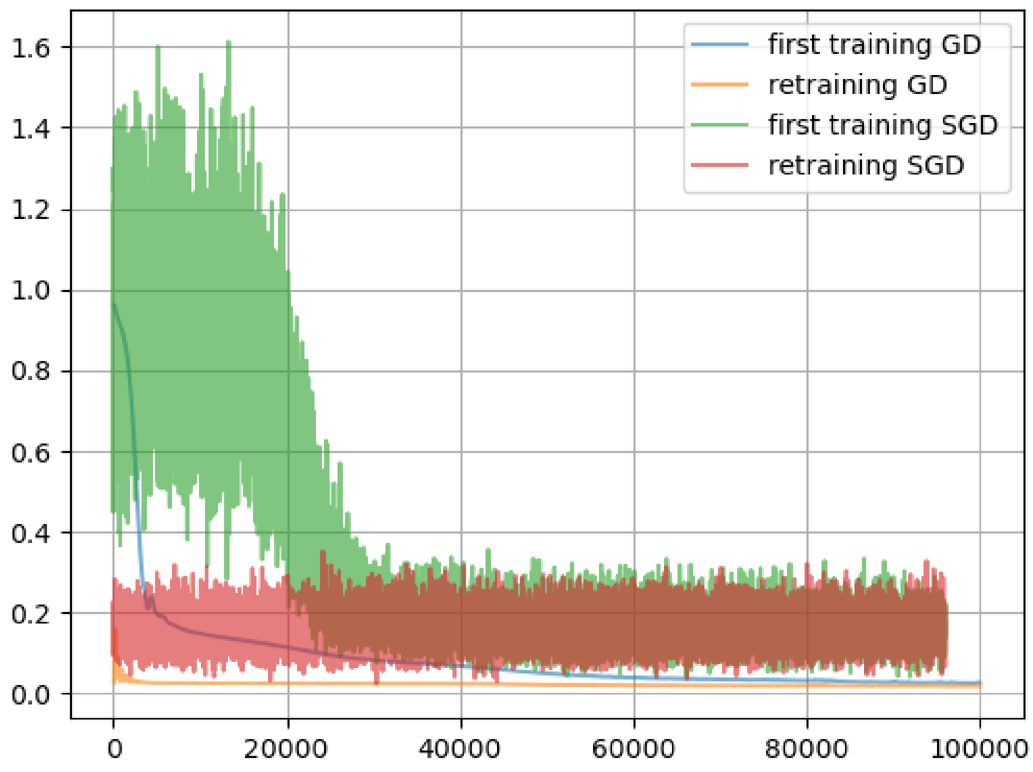
```python
        nn.init.normal_(parameter.weight,mean=0.0,std=1)
        nn.init.zeros_(parameter.bias)

retrain_trainer = pl.Trainer(max_epochs=max_epochs_run, enable_checkpointing=False)
retrain_trainer.fit(autoencoder_module_big_retrain, data_loader)
plt.plot(num_training_examples[1],autoencoder_module_big_retrain.loss_curve,label="retraining GD",alpha=0.6)


## now the same task for SGD ########################################################

dataset = TensorDataset(data, data)
data_loader = DataLoader(dataset, batch_size=8, shuffle=True, drop_last=True)

autoencoder_module_big_retrain = AutoencoderModuleSGD(hidded_channels=hidded_channels_big,latent_dim=1, input_dim=2, activation=nn.ReLU)

# start the training using a PyTorch Lightning Trainer
retrain_trainer = pl.Trainer(max_epochs=max_epochs_run, enable_checkpointing=False)

retrain_trainer.fit(autoencoder_module_big_retrain, data_loader)
plt.plot(num_training_examples[0],autoencoder_module_big_retrain.loss_curve,label="first training SGD",alpha = 0.6)

#now fix decoder parameters
for parameter in autoencoder_module_big_retrain.autoencoder.decoder.parameters():
    parameter.requires_grad = False

# reinitialize parameters for the encoder
for parameter in autoencoder_module_big_retrain.autoencoder.encoder.parameters():
    if isinstance(parameter,nn.Linear):
        nn.init.normal_(parameter.weight,mean=0.0,std=1)
        nn.init.zeros_(parameter.bias)

retrain_trainer = pl.Trainer(max_epochs=max_epochs_run, enable_checkpointing=False)
retrain_trainer.fit(autoencoder_module_big_retrain, data_loader)
plt.plot(num_training_examples[0],autoencoder_module_big_retrain.loss_curve,label="retraining SGD", alpha = 0.6)
```

```python
plt.legend()
plt.grid()
plt.show()
```

```
GPU available: False, used: False
TPU available: False, using: 0 TPU cores
HPU available: False, using: 0 HPUs

  | Name        | Type        | Params | Mode
----------------------------------------------------
0 | autoencoder | Autoencoder | 10.6 K | train
----------------------------------------------------
10.6 K    Trainable params
0         Non-trainable params
10.6 K    Total params
0.042     Total estimated model params size (MB)

Training: |                                               | 0/? [00:00<?, ?it/s]

`Trainer.fit` stopped: `max_epochs=1000` reached.
GPU available: False, used: False
TPU available: False, using: 0 TPU cores
HPU available: False, using: 0 HPUs

  | Name        | Type        | Params | Mode
----------------------------------------------------
0 | autoencoder | Autoencoder | 10.6 K | train
----------------------------------------------------
5.3 K     Trainable params
5.3 K     Non-trainable params
10.6 K    Total params
0.042     Total estimated model params size (MB)

Training: |                                               | 0/? [00:00<?, ?it/s]

`Trainer.fit` stopped: `max_epochs=1000` reached.
GPU available: False, used: False
TPU available: False, using: 0 TPU cores
HPU available: False, using: 0 HPUs

  | Name        | Type        | Params | Mode
----------------------------------------------------
0 | autoencoder | Autoencoder | 10.6 K | train
----------------------------------------------------
10.6 K    Trainable params
0         Non-trainable params
10.6 K    Total params
0.042     Total estimated model params size (MB)
```

```
Training: |                                            | 0/? [00:00<?, ?it/s]

`Trainer.fit` stopped: `max_epochs=1000` reached.
GPU available: False, used: False
TPU available: False, using: 0 TPU cores
HPU available: False, using: 0 HPUs

  | Name        | Type        | Params | Mode
----------------------------------------------------
0 | autoencoder | Autoencoder | 10.6 K | train
----------------------------------------------------
5.3 K     Trainable params
5.3 K     Non-trainable params
10.6 K    Total params
0.042     Total estimated model params size (MB)

Training: |                                            | 0/? [00:00<?, ?it/s]

IOPub message rate exceeded.
The Jupyter server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--ServerApp.iopub_msg_rate_limit`.

Current values:
ServerApp.iopub_msg_rate_limit=1000.0 (msgs/sec)
ServerApp.rate_limit_window=3.0 (secs)

IOPub message rate exceeded.
The Jupyter server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--ServerApp.iopub_msg_rate_limit`.

Current values:
ServerApp.iopub_msg_rate_limit=1000.0 (msgs/sec)
ServerApp.rate_limit_window=3.0 (secs)

IOPub message rate exceeded.
The Jupyter server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--ServerApp.iopub_msg_rate_limit`.

Current values:
ServerApp.iopub_msg_rate_limit=1000.0 (msgs/sec)
ServerApp.rate_limit_window=3.0 (secs)

IOPub message rate exceeded.
```

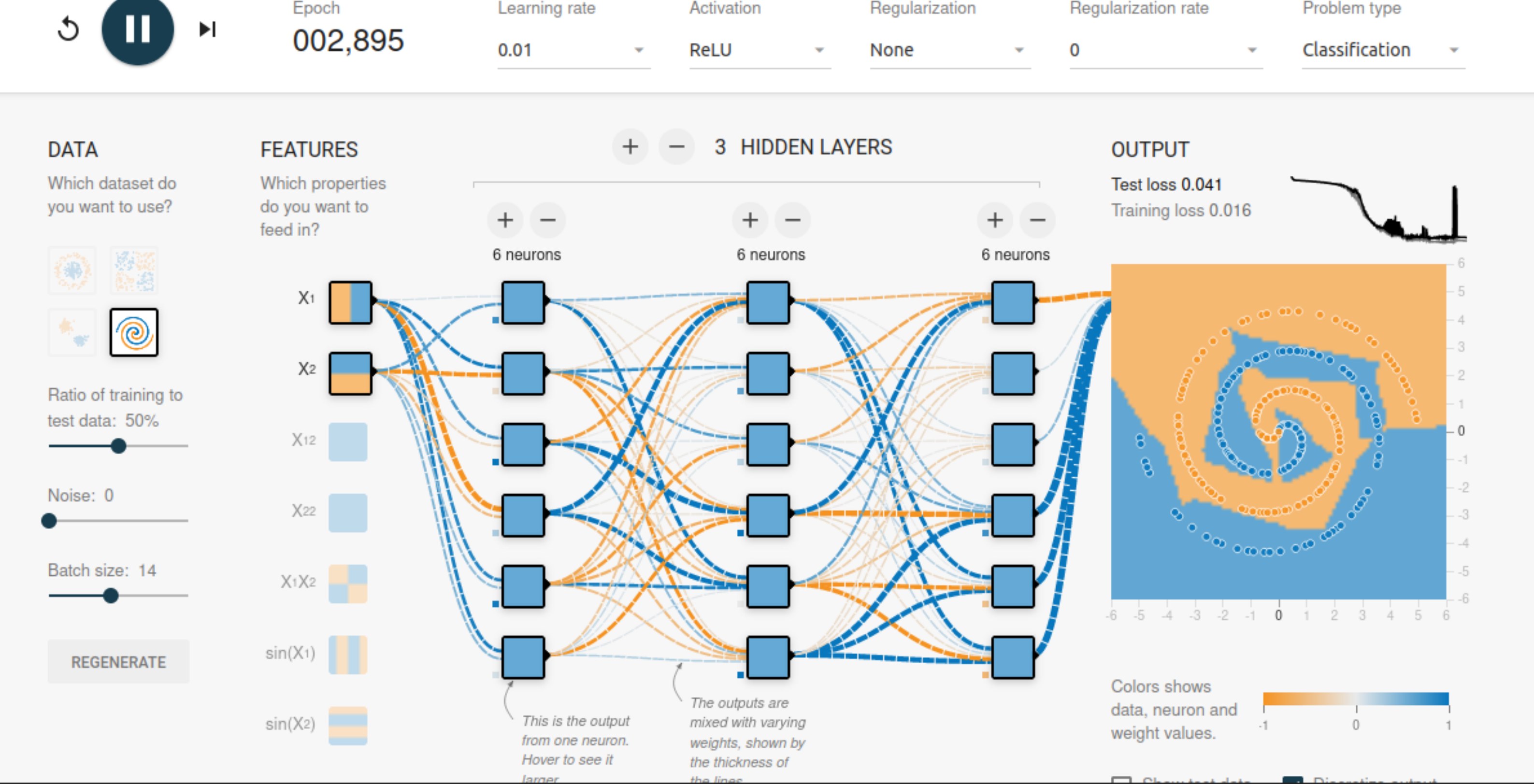
```
The Jupyter server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--ServerApp.iopub_msg_rate_limit`.

Current values:
ServerApp.iopub_msg_rate_limit=1000.0 (msgs/sec)
ServerApp.rate_limit_window=3.0 (secs)

IOPub message rate exceeded.
The Jupyter server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--ServerApp.iopub_msg_rate_limit`.

Current values:
ServerApp.iopub_msg_rate_limit=1000.0 (msgs/sec)
ServerApp.rate_limit_window=3.0 (secs)

`Trainer.fit` stopped: `max_epochs=1000` reached.
```



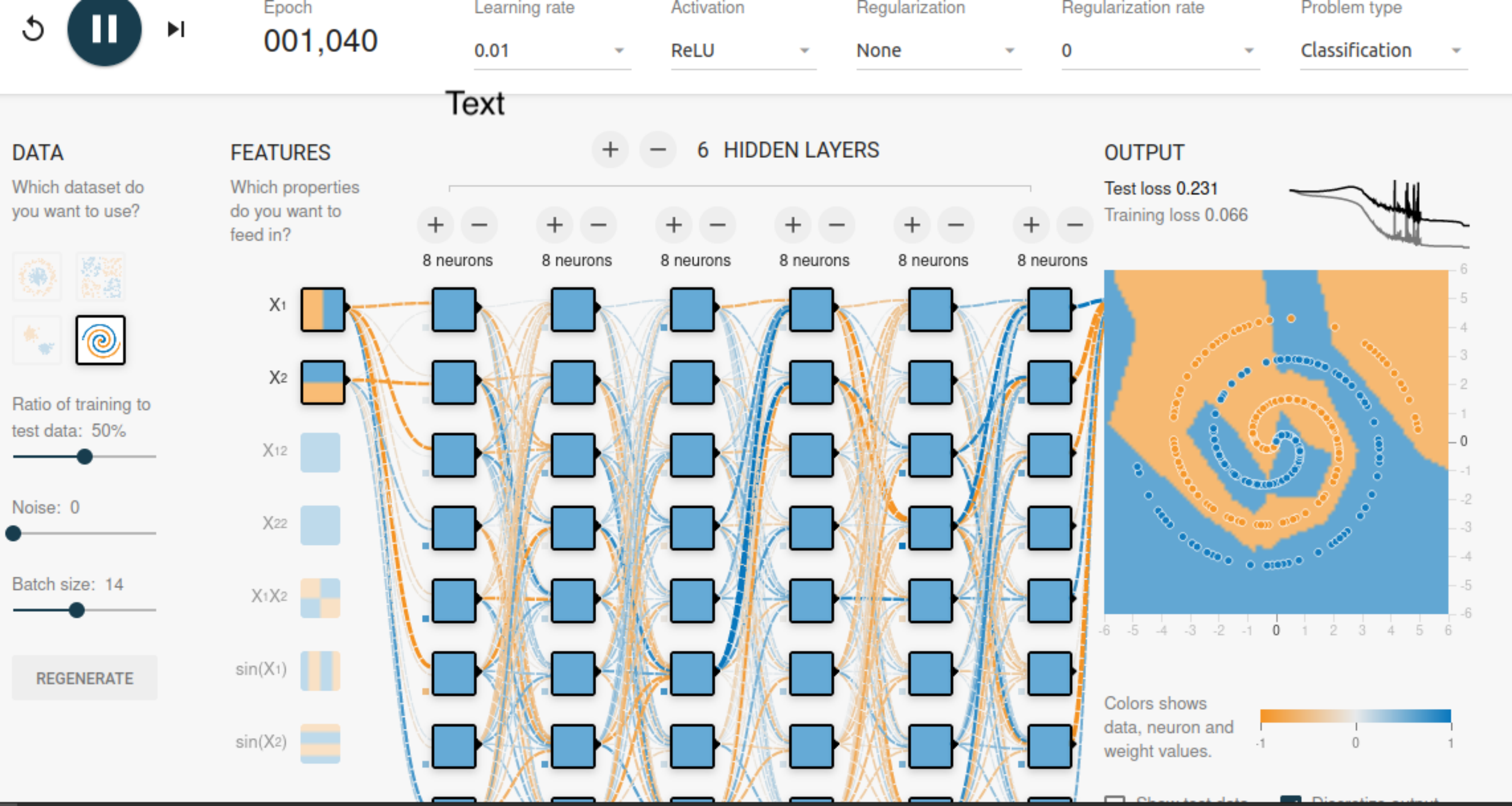

We see the gradient descent algorithm trains the network faster, when comapring to the examples seen, while in return taking more computing time.

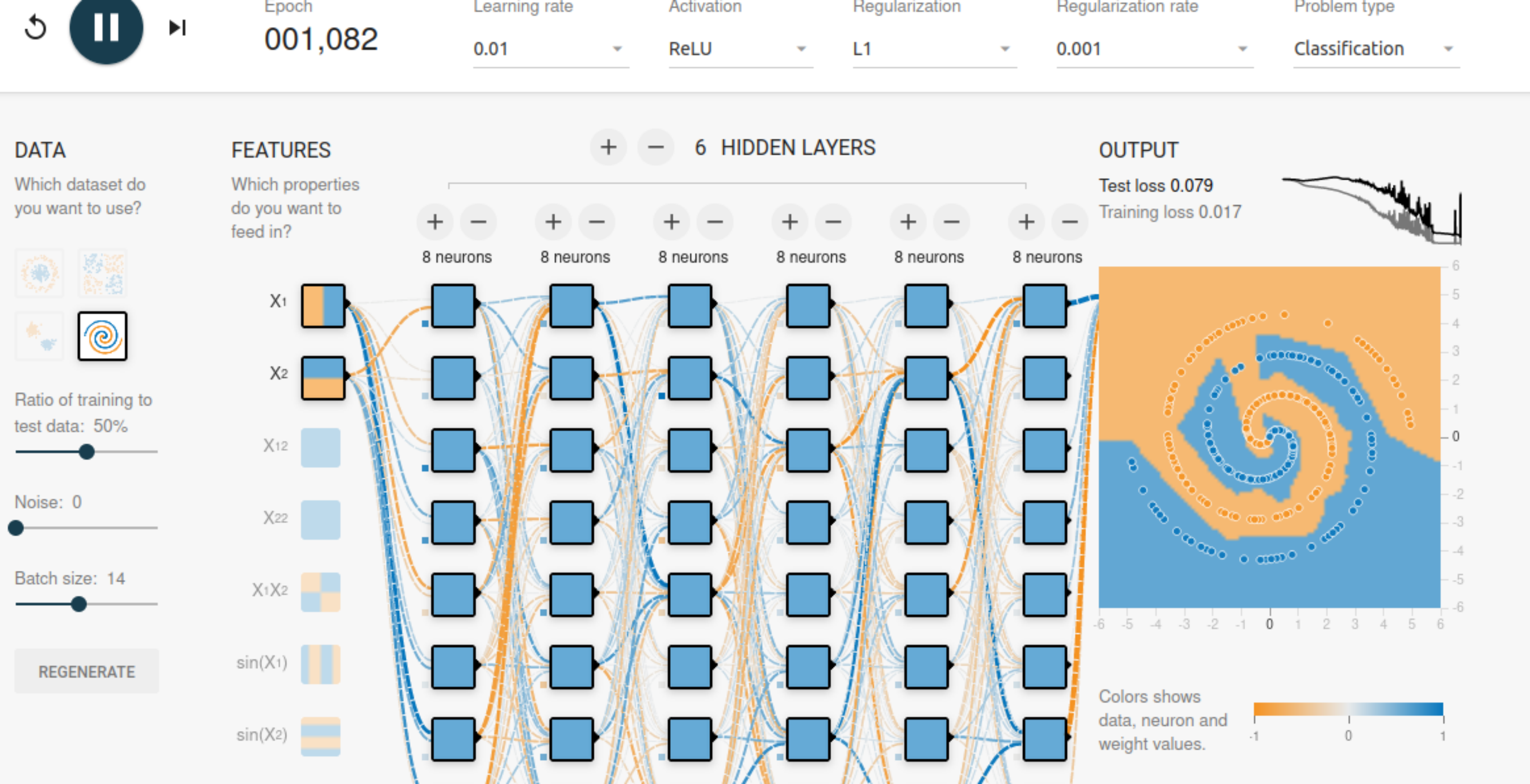
[ ]:

```

```



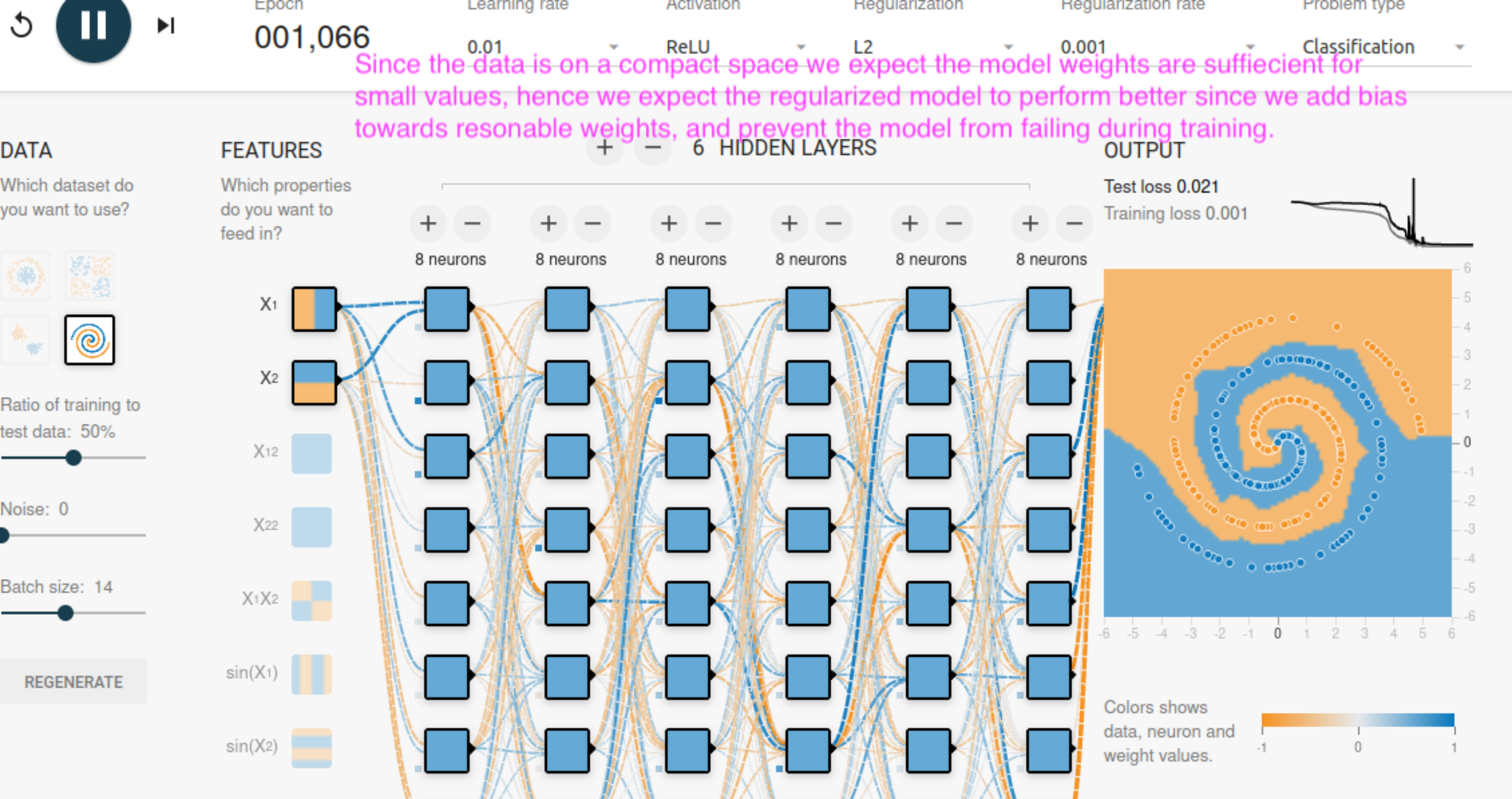
Epoch
002,895
Learning rate
0.01
Activation
ReLU
Regularization
None
Regularization rate
0
Problem type
Classification
DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Noise: 0
Batch size: 14
REGENERATE
FEATURES
Which properties do you want to feed in?
X1
X2
X12
X22
X1X2
sin(X1)
sin(X2)
3 HIDDEN LAYERS
6 neurons
6 neurons
6 neurons
This is the output from one neuron. Hover to see it larger.
The outputs are mixed with varying weights, shown by the thickness of the lines.
OUTPUT
Test loss 0.041
Training loss 0.016
Colors shows data, neuron and weight values.
-1
0
1

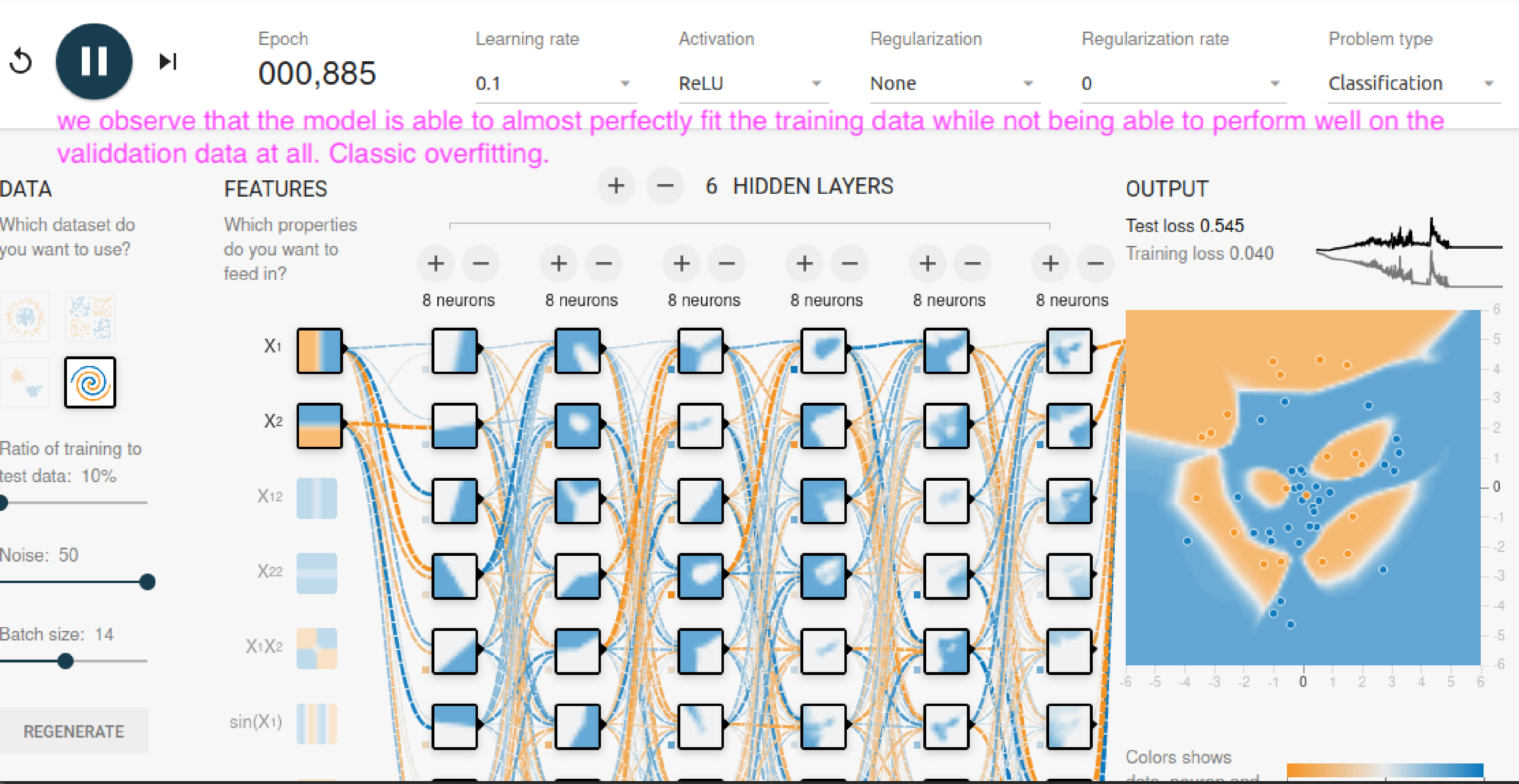

Epoch
001,040
Learning rate
0.01
Activation
ReLU
Regularization
None
Regularization rate
0
Problem type
Classification
Text
DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Noise: 0
Batch size: 14
REGENERATE
FEATURES
Which properties do you want to feed in?
X1
X2
X12
X22
X1X2
sin(X1)
sin(X2)
6 HIDDEN LAYERS
8 neurons
8 neurons
8 neurons
8 neurons
8 neurons
8 neurons
OUTPUT
Test loss 0.231
Training loss 0.066
Colors shows data, neuron and weight values.
-1
0
1



Epoch
001,082
Learning rate
0.01
Activation
ReLU
Regularization
L1
Regularization rate
0.001
Problem type
Classification
DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Noise: 0
Batch size: 14
REGENERATE
FEATURES
Which properties do you want to feed in?
X1
X2
X12
X22
X1X2
sin(X1)
sin(X2)
6 HIDDEN LAYERS
8 neurons
8 neurons
8 neurons
8 neurons
8 neurons
8 neurons
OUTPUT
Test loss 0.079
Training loss 0.017
Colors shows data, neuron and weight values.

Since the data is on a compact space we expect the model weights are suffiecient for small values, hence we expect the regularized model to perform better since we add bias towards resonable weights, and prevent the model from failing during training.




Epoch
000,885
Learning rate
0.1
Activation
ReLU
Regularization
None
Regularization rate
0
Problem type
Classification
we observe that the model is able to almost perfectly fit the training data while not being able to perform well on the validdation data at all. Classic overfitting.
DATA
Which dataset do you want to use?
Ratio of training to test data: 10%
Noise: 50
Batch size: 14
REGENERATE
FEATURES
Which properties do you want to feed in?
6 HIDDEN LAYERS
8 neurons
8 neurons
8 neurons
8 neurons
8 neurons
8 neurons
OUTPUT
Test loss 0.545
Training loss 0.040
Colors shows