Using Generative Adversarial Networks for Data Generation

Team DowGAN: Daniel Kuo, Emily Miura-Stempel, Emily Nishiwaki, Arty Timchenko

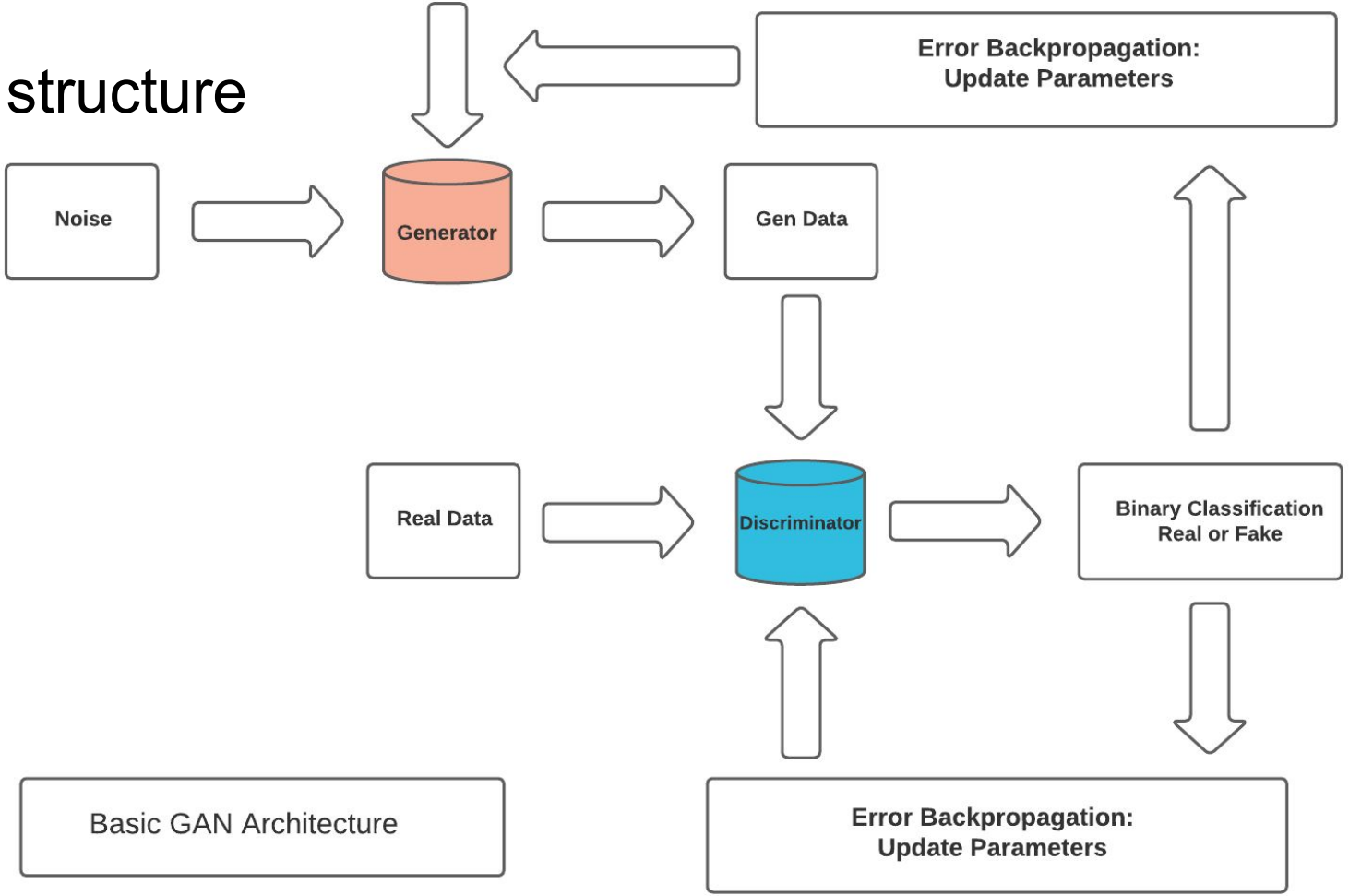
Motivation

Problem: Obtaining large amounts of experimental data is expensive and resource consuming, especially on the industrial level scale

Project: Exploring whether generative adversarial networks can resolve this problem by:

- Generating synthetic time series data
- Increasing resolution of data
- Generating points beyond training data

GAN structure



Basic GAN Architecture

Our GAN

Discriminator

```
class Discriminator(nn.Module):
    ''' Classifies data as real or synthetic, used to train generator'''
    # Consists of 3 sequential linear layers, standerdized by dropout
    # Uses a Relu activation function
    def __init__(self,df_dim,batch_size,drop_out):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(df_dim, (batch_size*8)),
            nn.ReLU(),
            nn.Dropout(drop_out),
            nn.Linear((batch_size*8), (batch_size*4)),
            nn.ReLU(),
            nn.Dropout(drop_out),
            nn.Linear((batch_size*4), (batch_size*2)),
            nn.ReLU(),
            nn.Dropout(drop_out),
            nn.Linear((batch_size*2), 1),
            nn.Sigmoid(),
        )
    def forward(self, x):
        output = self.model(x)
        return output
```

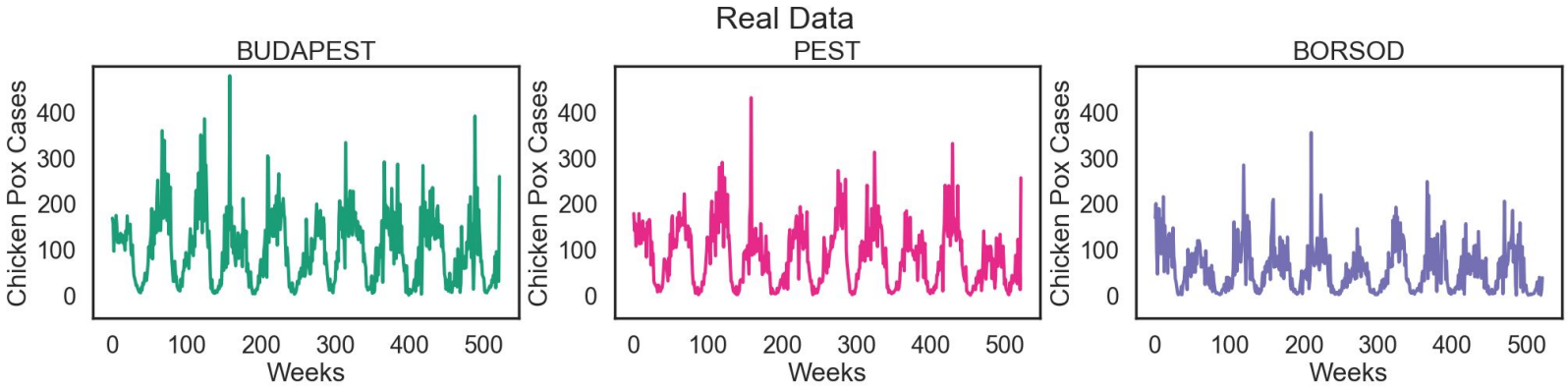
Generator

```
class Generator(nn.Module):
    ''' Generates synthetic data'''
    # Consists of 3 sequential linear layers, standerdized by dropout
    # Uses a Relu activation function
    def __init__(self,df_dim,batch_size,drop_out):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(df_dim, (batch_size)),
            nn.ReLU(),
            nn.Linear((batch_size), (batch_size*2)),
            nn.ReLU(),
            nn.Linear((batch_size*2), df_dim),
        )
    def forward(self, x):
        output = self.model(x)
        return output
```

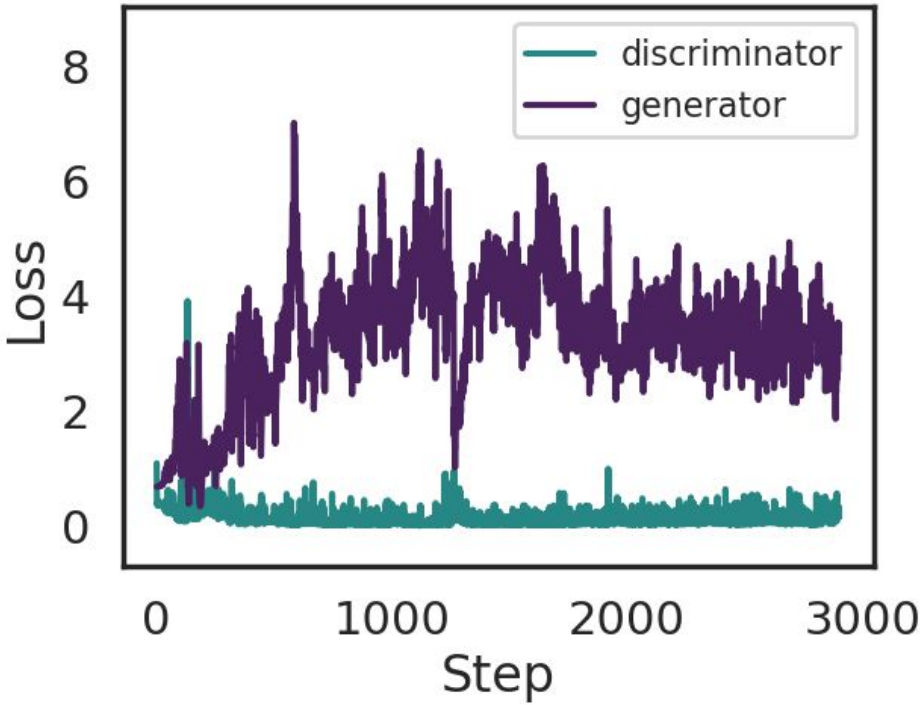
Data Visualization: Hungary Chicken Pox Dataset

	BUDAPEST	BARANYA	BACS	BEKES	BORSOD	CSONGRAD	FEJER	GYOR	HAJDU	HEVES	JASZ	KOMAROM	NOGRAD	PEST	SOMOGY	SZABOLCS	TOLNA	VAS	VESZPREM	ZALA
0	168	79	30	173	169	42	136	120	162	36	130	57	2	178	66	64	11	29	87	68
1	157	60	30	92	200	53	51	70	84	28	80	50	29	141	48	29	58	53	68	26
2	96	44	31	86	93	30	93	84	191	51	64	46	4	157	33	33	24	18	62	44
3	163	49	43	126	46	39	52	114	107	42	63	54	14	107	66	50	25	21	43	31
4	122	78	53	87	103	34	95	131	172	40	61	49	11	124	63	56	7	47	85	60
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
517	95	12	41	6	39	0	16	15	14	10	56	7	13	122	4	23	4	11	110	10
518	43	39	31	10	34	3	2	30	25	19	34	20	18	70	36	5	23	22	63	9
519	35	7	15	0	0	0	7	7	4	2	30	36	4	72	5	21	14	0	17	10
520	30	23	8	0	11	4	1	9	10	17	27	17	21	12	5	17	1	1	83	2
521	259	42	49	32	38	15	11	98	61	38	112	61	53	256	45	39	27	11	103	25

522 rows × 20 columns



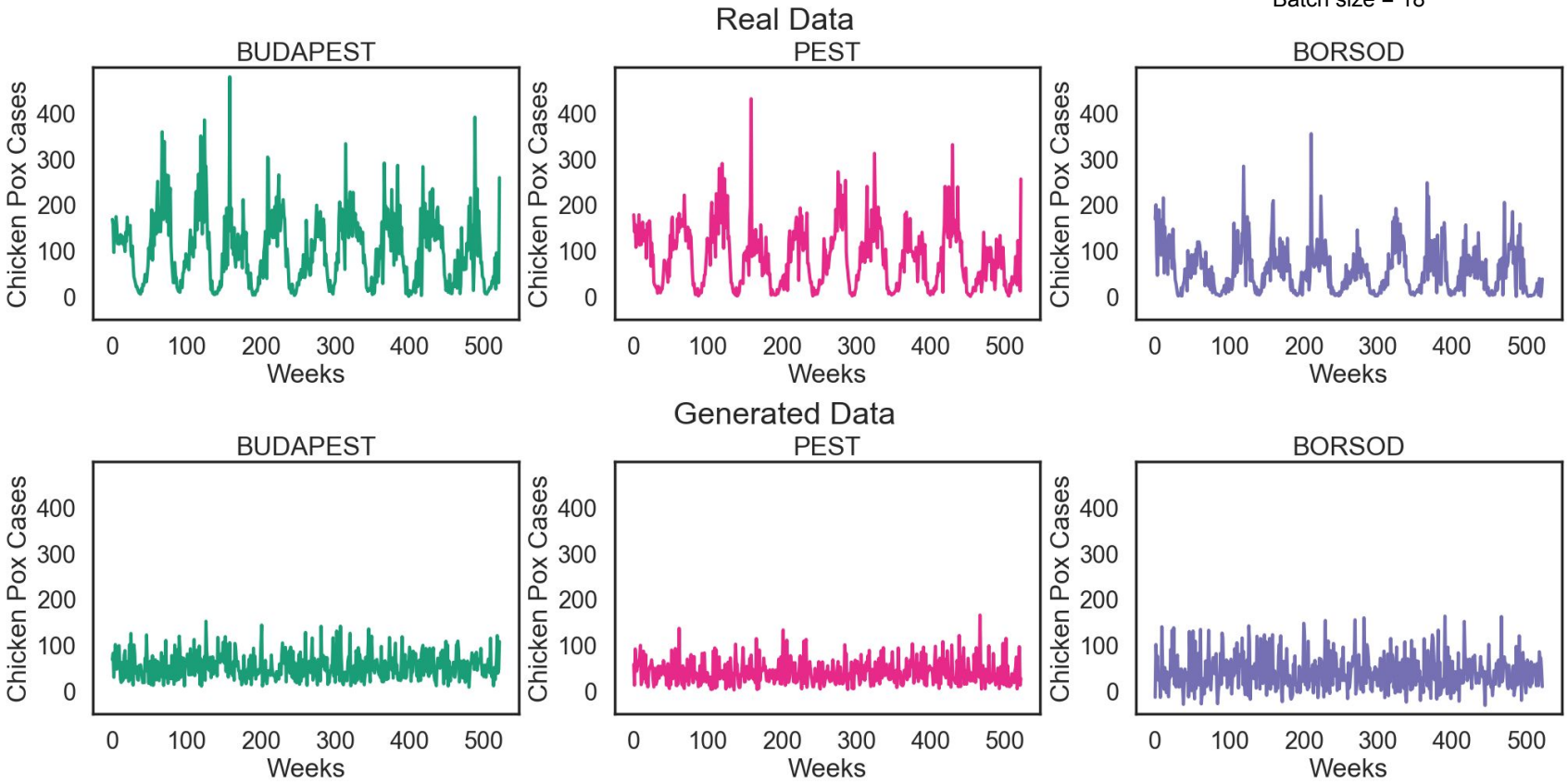
Generator Loss is not decreasing



Parameters:
Num_data = 522
Num_epochs = 100
Learning rate = 0.002
Drop out = 0.2
Batch size = 18

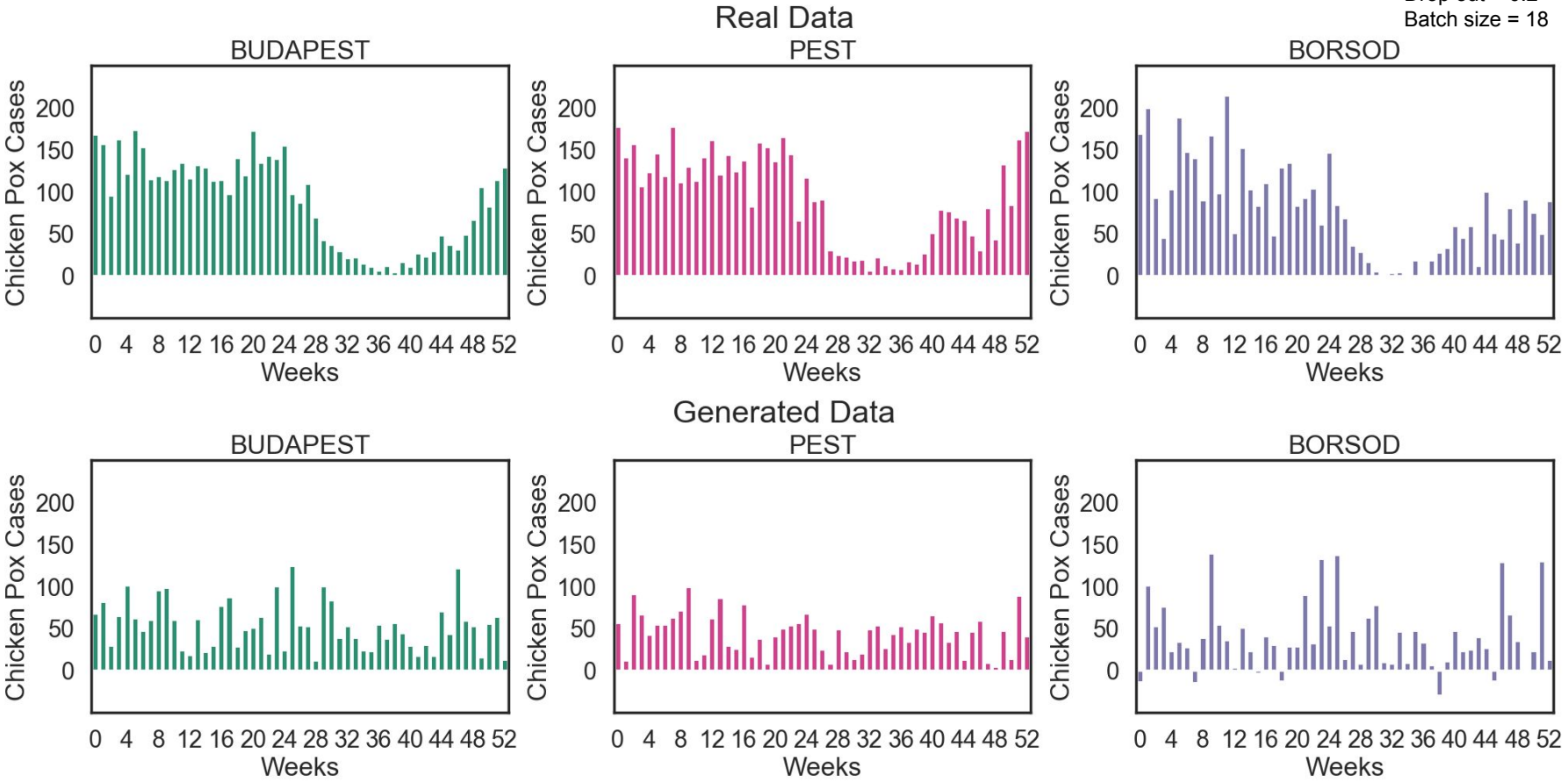
Generated Data vs. Real Data Visualization

Parameters:
Num_data = 522
Num_epochs = 100
Learning rate = 0.002
Drop out = 0.2
Batch size = 18



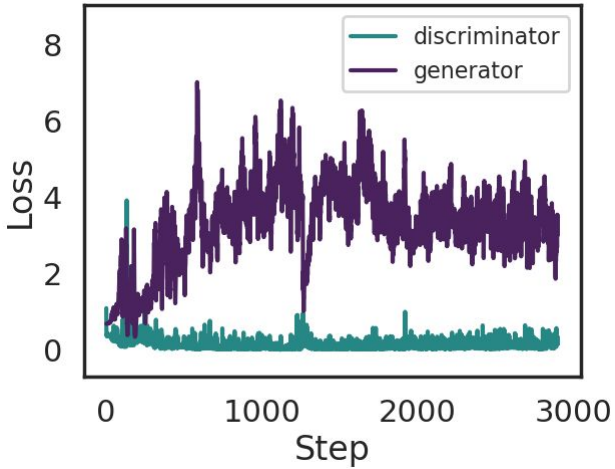
Generated Data vs. Real Data: 52 week subset

Parameters:
Num_data = 522
Num_epochs = 100
Learning rate = 0.002
Drop out = 0.2
Batch size = 18

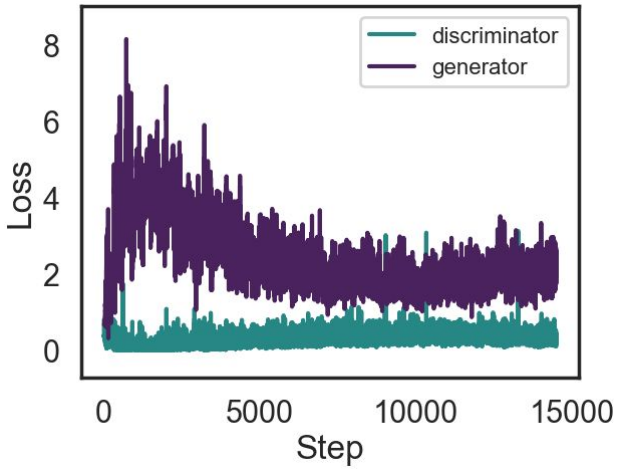


Impact of epochs on loss

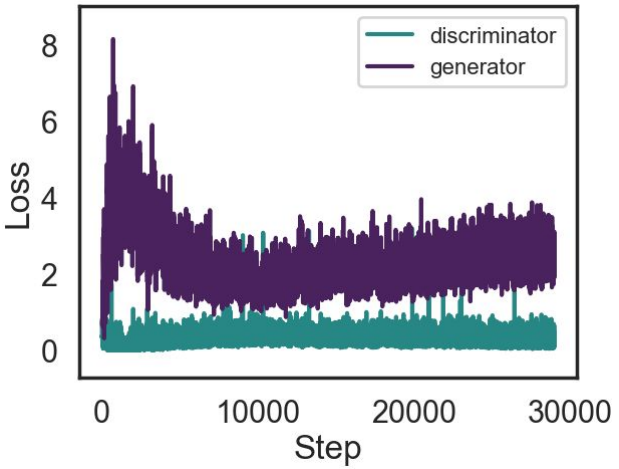
Parameters:
Num_data = 522
Learning rate = 0.002
Drop out = 0.2
Batch size = 18



100 epochs



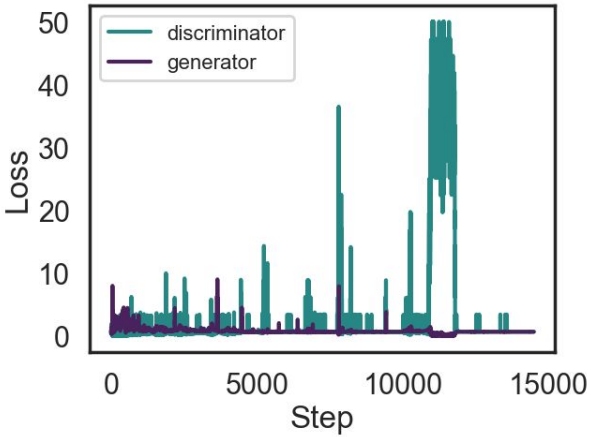
500 epochs



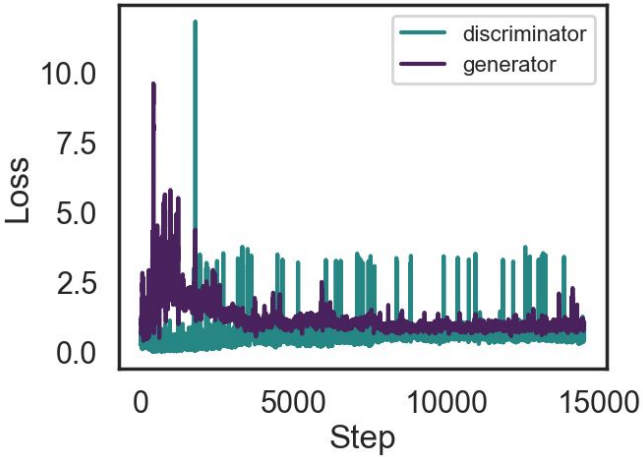
1000 epochs

Impact of learning rate on loss

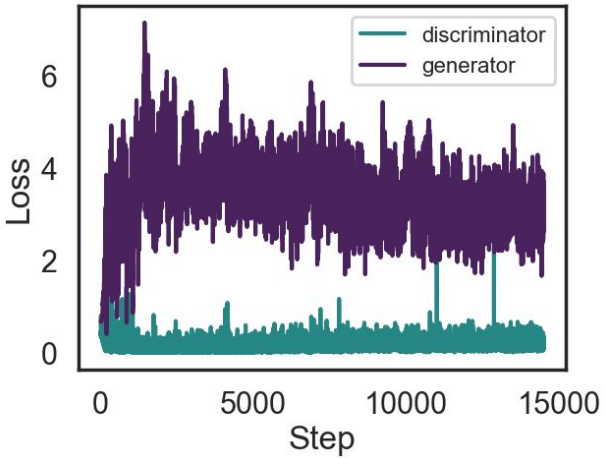
Parameters:
Num_data = 522
Num_epochs = 500
Drop out = 0.2
Batch size = 18



lr = 0.01



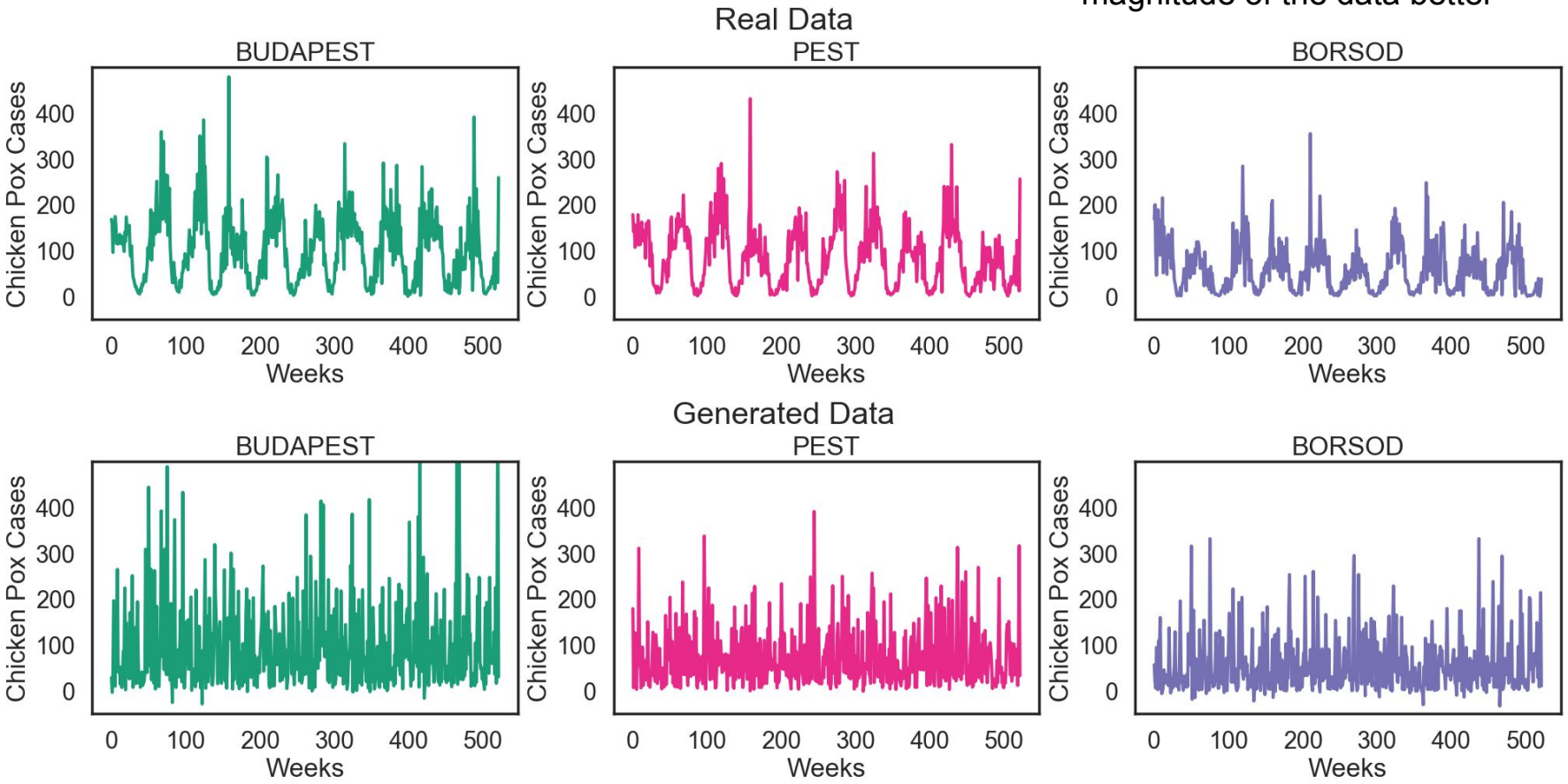
lr = 0.005



lr = 0.001

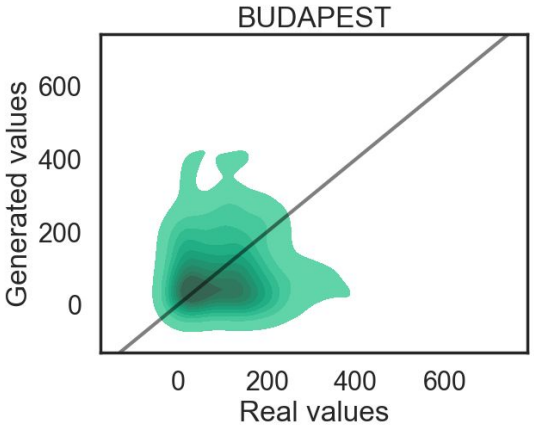
Using num_epoch = 500 and lr = 0.005

Using brief parameter optimization the generator appears to match the magnitude of the data better

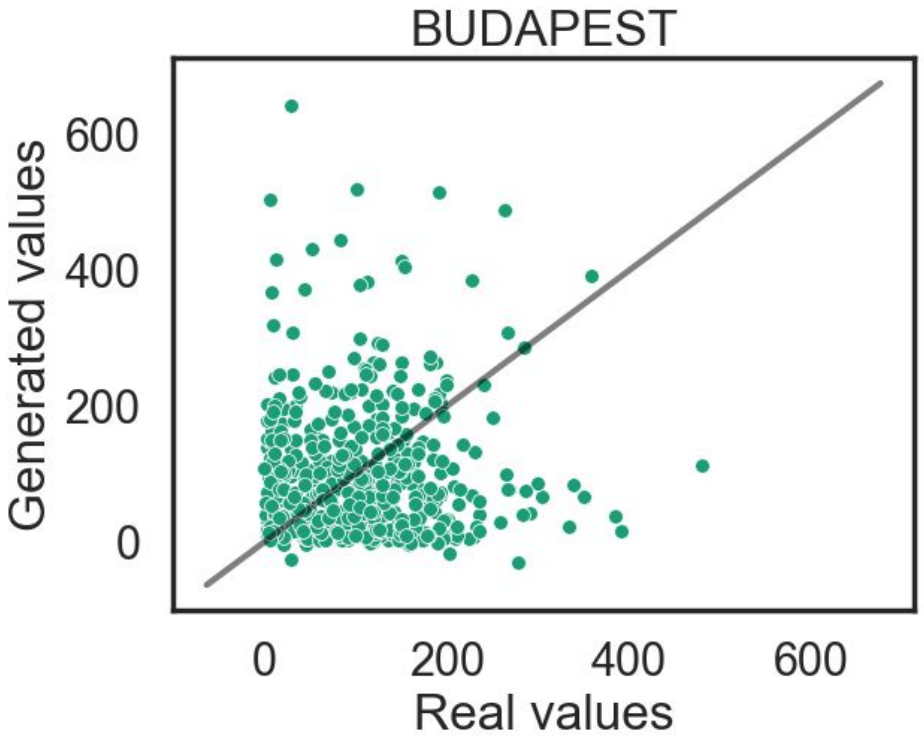


Conclusions

- Successfully written code that runs a GAN on time series data
- GAN does not yet have accurate predictions
    - Diminished gradient - discriminator is too successful and generator gradients vanish



Parity Plot
R^2 score = –1.48



New Model Hyperparameters: Loss Function

Wasserstein Loss:

- Seeks to minimize distance between distributions of data in the training set and generated set.
- Provides a continuous gradient
- More sensitive to hyperparameter optimization and model architecture

Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

Require: : α, the learning rate. c, the clipping parameter. m, the batch size. n_{critic}, the number of iterations of the critic per generator iteration.

Require: : w_0, initial critic parameters. θ_0, initial generator's parameters.

while θ has not converged **do**
  for $t = 0, ..., n_{\text{critic}}$ **do**
    Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_r$ a batch from the real data.
    Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
    $g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))\right]$
    $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
    $w \leftarrow \text{clip}(w, -c, c)$
  end for
  Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
  $g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))$
  $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
end while

WGAN https://arxiv.org/pdf/1701.07875.pdf

Generator and Discriminator Architecture

Gated Recurrent Unit

- Used to model sequential data
- Addresses the issue of vanishing gradients occurring in RNN's
- Weights are calculate by backpropagation using gradient descent
- Consists of:
  - An Input Gate which determines the importance of each input element
  - An Update Gate which determines how the hidden state is retained and updated
  - A Candidate State that introduces new information to the hidden state
  - Hidden State which is the output and a combination of previous hidden state and candidate state

```python
class Generator(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers):
        super().__init__()
        self.rnn = nn.GRU(input_size, hidden_size, num_layers)
        self.fc1 = nn.Linear(hidden_size, hidden_size)
        self.norm1 = LayerNorm(hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)
        self.norm2 = LayerNorm(output_size)
        self.dropout = nn.Dropout(0.1)
        self.leakyrelu = nn.LeakyReLU()

    def forward(self, x):
        with torch.backends.cudnn.flags(enabled=False):
            x, _ = self.rnn(x)
            x = self.fc1(x)
            x = self.norm1(x)
            x = self.leakyrelu(x)
            x = self.dropout(x)
            x = self.fc2(x)
            x = self.norm2(x)
            return x
```

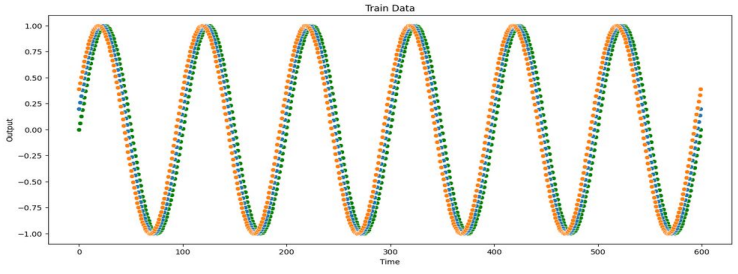
Transfer Learning

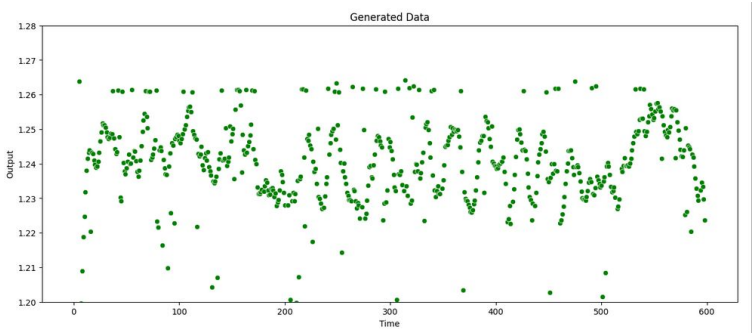
To speed up the process of training the GAN, the weights and bias from similar models are imported into the new model.

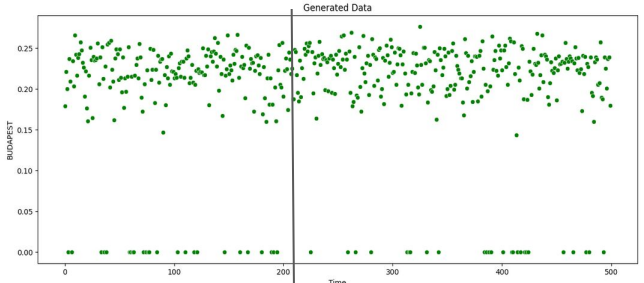
Helps capture temporal dependencies in new data by using cleaner data with similar shape

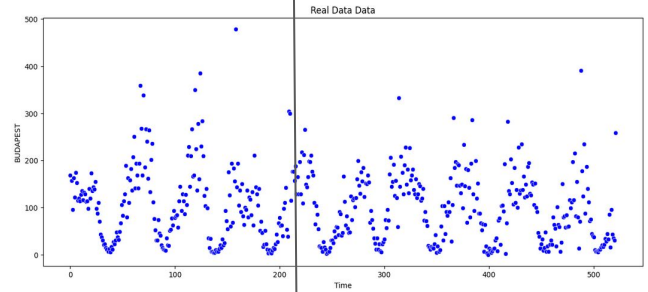
Lower training time by saving results of previous successful runs.

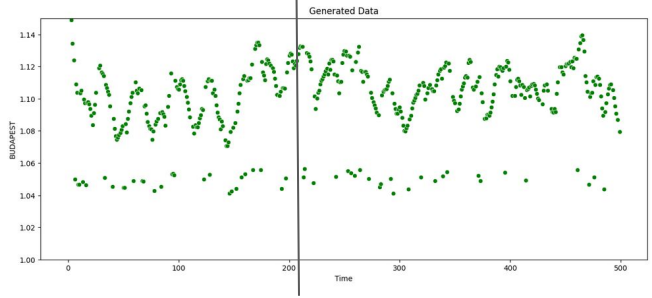
Transfer Learning











Model trained on 200 points no transfer learning

Model trained on first 200 points with transfer learning

Next Steps

Generator/Discriminator Architecture:

- Convolutional Layers, GRU, Fully Connected
- Node, number of layers, activation function optimization, optuna
- Standardization: Layernorm, Batch Norm, Dropout

Dataloader:

- Batch size

Training:

- Number of epochs, learning rate, training set size, transfer learning
- Structured Input noise tensor for training