


Podstawy Inżynierii Danych		
Projekt ZTSI		
Dominik Ossowski 28.11.2023		
Sprawozdanie z projektu		

Celem projektu jest wytrenowanie modelu, którego zadanie polega na przewidywaniu zachowania roju na podstawie dostarczonych parametrów. Projekt realizuję w jupyter notebook za pośrednictwem google colab. Często w tego typu projektach korzystam z powyższego rozwiązania, ponieważ jest to wygodne, a zasoby udostępnione przez google – wystarczające.

```
df_initial = pd.read_csv('/content/drive/MyDrive/ztsi202324.csv', names=["Z", "O", "A", "B", "C", "D", "E", "F", "P", "Q", "R", "S", "T"], nrows=3000, usecols=["A", "B", "C", "D", "E", "F", "P", "Q", "R", "S", "T"], df_initial
```

Załadowałem pierwszych 3000 rekordów i nadałem nazwy kolumnom zgodnie z instrukcją.

## 1. Analiza danych

W analizowaniu danych pomogło mi narzędzie ProfileReport z pakietu ydata\_profiling. Tworzy ono specjalny raport, który zawiera informacje na temat każdej zmiennej oraz ich histogram.

- Zmienna P zawsze ma wartość 0 poza 2 przypadkami



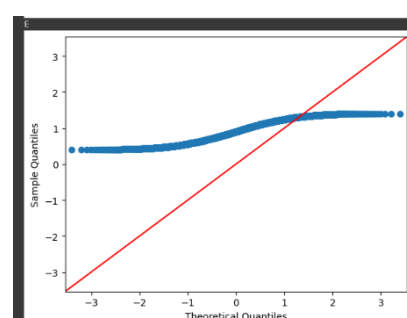
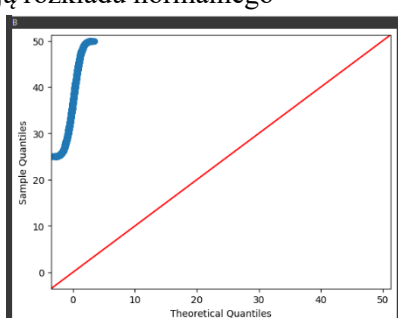
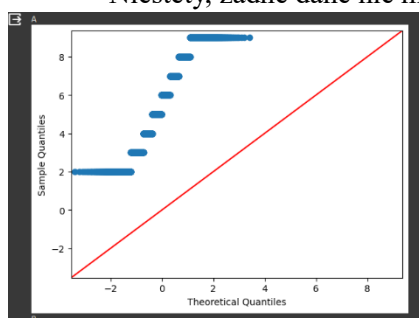
```
[ ] df_initial.loc[df_initial['P'] != 0]
```

	A	B	C	D	E	F	P	Q	R	S	T
274	8	28.196374	26.935640	27.548323	0.470696	0.414093	0.000379	0.999621	0.000000	0.000000	0.000000
1566	7	30.084534	21.894707	26.735059	0.534062	0.462471	0.000354	0.076923	0.194257	0.728465	5116.341707

Pierwszy postanowiłem usunąć z racji na to, że nie nastąpiło przejście

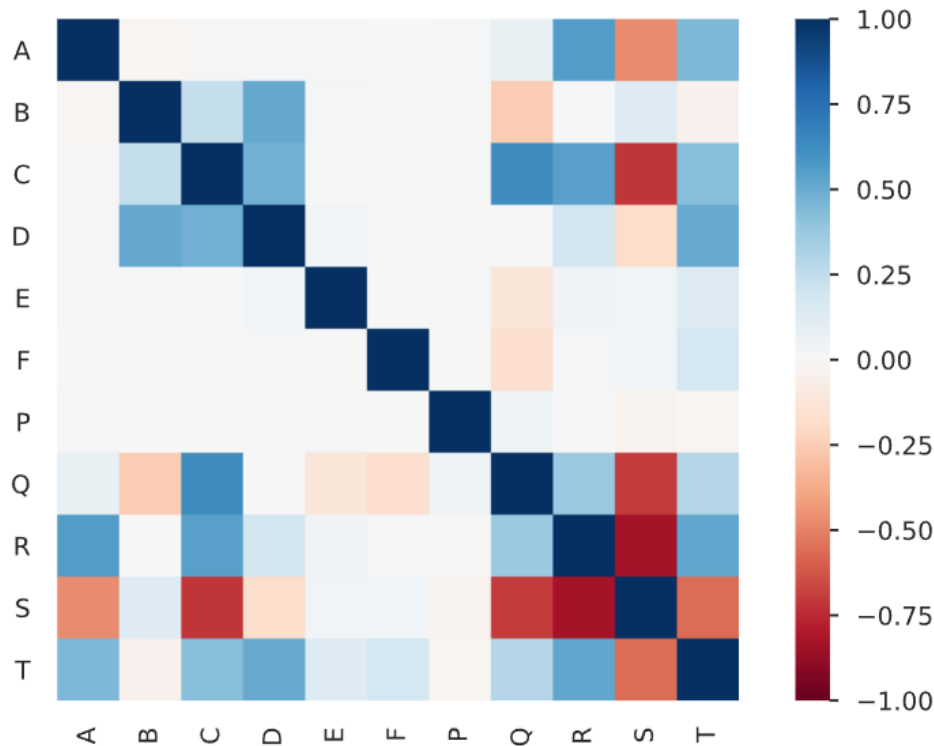
W drugim przypadku zmieniłem wartość p na 0 i zachowałem obserwację

- Żadnych pustych wartości i duplikatów nie ma
- Niestety, żadne dane nie mają rozkładu normalnego



Do wizualizacji rozkładu normalnego wykorzystałem wykres kwantylowy (qq plot), gdzie warunkiem występowania rozkładu normalnego jest ułożenie się niebieskich kropek mniej więcej wzdłuż narysowanej linii. Znajdują się tu tylko 3 przykłady („A”, „B” i „E”), reszta jest w kodzie.

- Wszystkie wartości w kolumnach mają kilka miejsc po przecinku, dlatego postanowiłem zaokrąglić wszystkie liczby do 3 miejsca po przecinku, aby uzyskać większą klarowność.
- Dużym problemem w trakcie realizacji projektu okazała się kolumna „T” – znajdujące się w niej wartości są nieproporcjonalnie duże w porównaniu do reszty danych i zawierają wiele obserwacji odstających.



Patrząc na histogram widać kilka zależności:

- Im prawdopodobieństwo oscylacji i wibracji jest niższe to większa szansa na przejście.
- Im większa szansa na przejście tym mniejszy stopień zakłóceń przy przejściu
- Ilość dronów na duży wpływ na występowanie wibracji
- Promień wewnętrznej strefy reakcji ma duży wpływ na prawdopodobieństwo oscylacji
- Jeżeli prawdopodobieństwo wibracji jest wysokie to prawdopodobieństwo oscylacji jest niskie (na odwrót jest tak samo).

Mój pomysł na projekt opiera na odpowiedzeniu na pytanie czy jeden model sieci neuronowej produkujący cztery wyniki poradzi sobie lepiej niż kilka modeli wyspecjalizowanych w przewidywaniu jednej bądź więcej wartości (ale nie wszystkich na raz)?

Zdecydowałem się wykorzystać sieci, ponieważ one z reguły powinny dawać sobie radę z danymi, które są „chaotyczne”, nie mają rozkładu normalnego. Do dyspozycji mam 2999 obserwacji, co niestety nie jest dużą ilością, jeżeli bierzemy pod uwagę trenowanie sieci.

## 2. Przygotowanie danych do wprowadzenia do modeli

Pierwszym problemem okazały się wartości w kolumnie „T”. Są one na tyle nieproporcjonalnie duże, że wprowadzenie ich na surowo do modelu i potem próba uczenia, kończyła się fiaskiem. Próbowałem zwiększać ilość epok do 100.000, tworzyć większe, bardziej złożone sieci, ale nic nie przynosiło skutku. Jedynym wyjściem z sytuacji okazało się zastosowanie normalizacji (konkretnie standaryzacji) target data. Nie spowodowało to drastycznej poprawy wyniku, ale przynajmniej  $r^2$  dla targetu „T” rzadziej wychodził lekko ujemny.

```
scaler = StandardScaler()
target_scaler = MinMaxScaler()
```

Kolejny problem pojawił się, kiedy testowałem działanie metody `inverse_transform()`, której zadaniem było odwrócenie standaryzacji, aby rozszyfrować wynik predykcji.

```
labels_train_s
array([[0.      , 0.      , 1.      , 0.01557233],
       [0.002   , 0.65631929, 0.406   , 0.02032364],
       [0.      , 0.      , 1.      , 0.01059939],
       ...,
       [0.      , 0.      , 1.      , 0.00485121],
       [0.343   , 0.02217295, 0.636   , 0.02433822],
       [0.      , 0.      , 1.      , 0.00125552]])

[ ] labels_test_s_ada = target_scaler.inverse_transform(labels_train_s_ada)
labels_test_s_ada
array([[0.0000000e+00, 0.0000000e+00, 1.0000000e+00, 7.2028350e-01],
       [2.0000000e-03, 5.9200000e-01, 4.0600000e-01, 9.4005090e-01],
       [0.0000000e+00, 0.0000000e+00, 1.0000000e+00, 4.9026490e-01],
       ...,
       [0.0000000e+00, 0.0000000e+00, 1.0000000e+00, 2.2438800e-01],
       [3.4300000e-01, 2.0000000e-02, 6.3600000e-01, 1.1257415e-01],
       [0.0000000e+00, 0.0000000e+00, 1.0000000e+00, 5.8073100e-01]])
```

Jak widać nie przywraca ona wartości sprzed standaryzacji.

W tej sytuacji zmuszony byłem napisać swoją standaryzację korzystając z ogólnego wzoru

$$z = \frac{x - \mu}{\sigma},$$

```

means = np.mean(features_train, axis=0)
std_devs = np.std(features_train, axis=0)

features_train_s_m = (features_train - means) / std_devs
features_test_s_m = (features_test - means) / std_devs

means_l = np.mean(labels_train, axis=0)
std_devs_l = np.std(labels_train, axis=0)

labels_train_s_m = (labels_train - means_l) / std_devs_l
labels_test_s_m = (labels_test - means_l) / std_devs_l

```

```

features_train_s_m

```

	A	B	C	D	E	F	P
170	-0.262508	0.571482	-0.074108	0.641345	0.619561	1.236161	0.0
2699	1.507686	0.870613	1.529470	2.355240	-1.262695	-1.269930	0.0
43	0.622589	-0.085826	-0.697233	0.000190	-0.962225	0.696440	0.0
87	-0.262508	-1.453934	-0.071297	-1.158426	0.567756	1.280851	0.0
1411	1.507686	-0.277281	-1.268007	0.872501	-1.183260	1.566181	0.0
...	...	...	...	...	...	...	...
1638	-0.705057	0.389476	-0.711463	0.347949	-1.217797	-0.173300	0.0
1095	-1.590153	0.885062	-0.498718	-0.873223	-0.630671	1.263662	0.0
1130	-0.262508	1.535841	-0.890302	-0.755991	1.255038	0.806447	0.0
1294	-0.262508	-0.822745	0.909858	0.287284	0.474506	-0.668330	0.0
860	-1.590153	1.723127	-0.431083	-0.682405	-1.083104	-0.709583	0.0

2549 rows x 7 columns

```

[ ] reversed_data = (features_train_s_m * std_devs) + means

[ ] reversed_data

```

	A	B	C	D	E	F	P
170	5.0	41.264	20.236	32.503	1.083	1.257	NaN
2699	9.0	43.417	29.364	43.380	0.538	0.528	NaN
43	7.0	36.533	16.689	28.434	0.625	1.100	NaN
87	5.0	26.686	20.252	21.081	1.068	1.270	NaN
1411	9.0	35.155	13.440	33.970	0.561	1.353	NaN

Odwracanie działa poprawnie.

Ostatnim krokiem była zamiana danych na tensory, aby można było wprowadzić je do modelu.

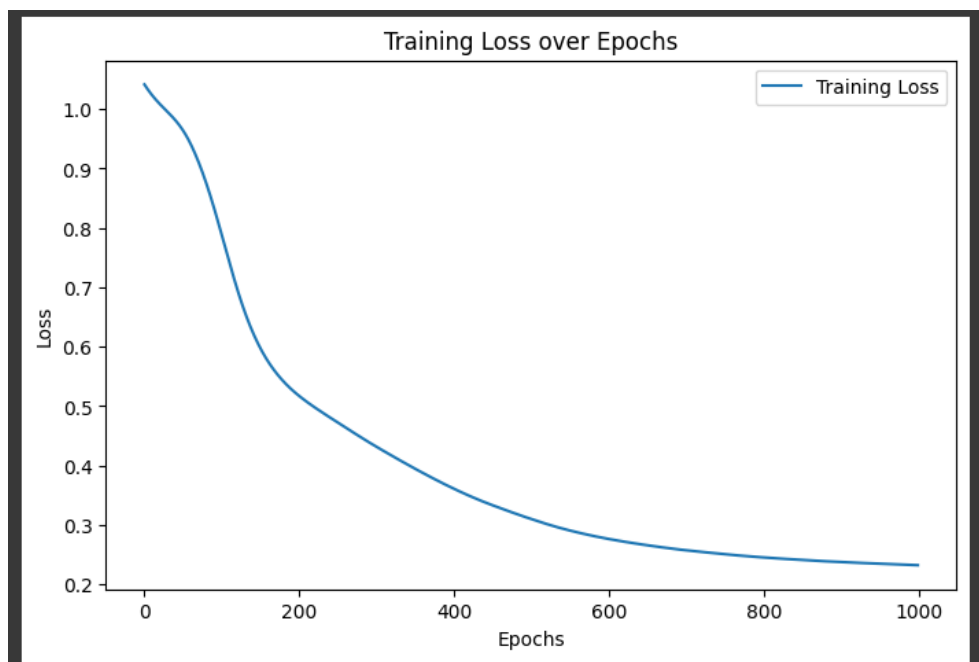
### 3. Model dający 4 wyniki

```
class NeuralNetwork(nn.Module):  
    def __init__(self):  
        super(NeuralNetwork, self).__init__()  
        self.fc1 = nn.Linear(7, 16)  
        self.fc2 = nn.Linear(16, 7)  
        self.fc3 = nn.Linear(7, 4)  
  
    def forward(self, x):  
        x = torch.relu(self.fc1(x))  
        x = torch.relu(self.fc2(x))  
        x = self.fc3(x)  
        return x
```

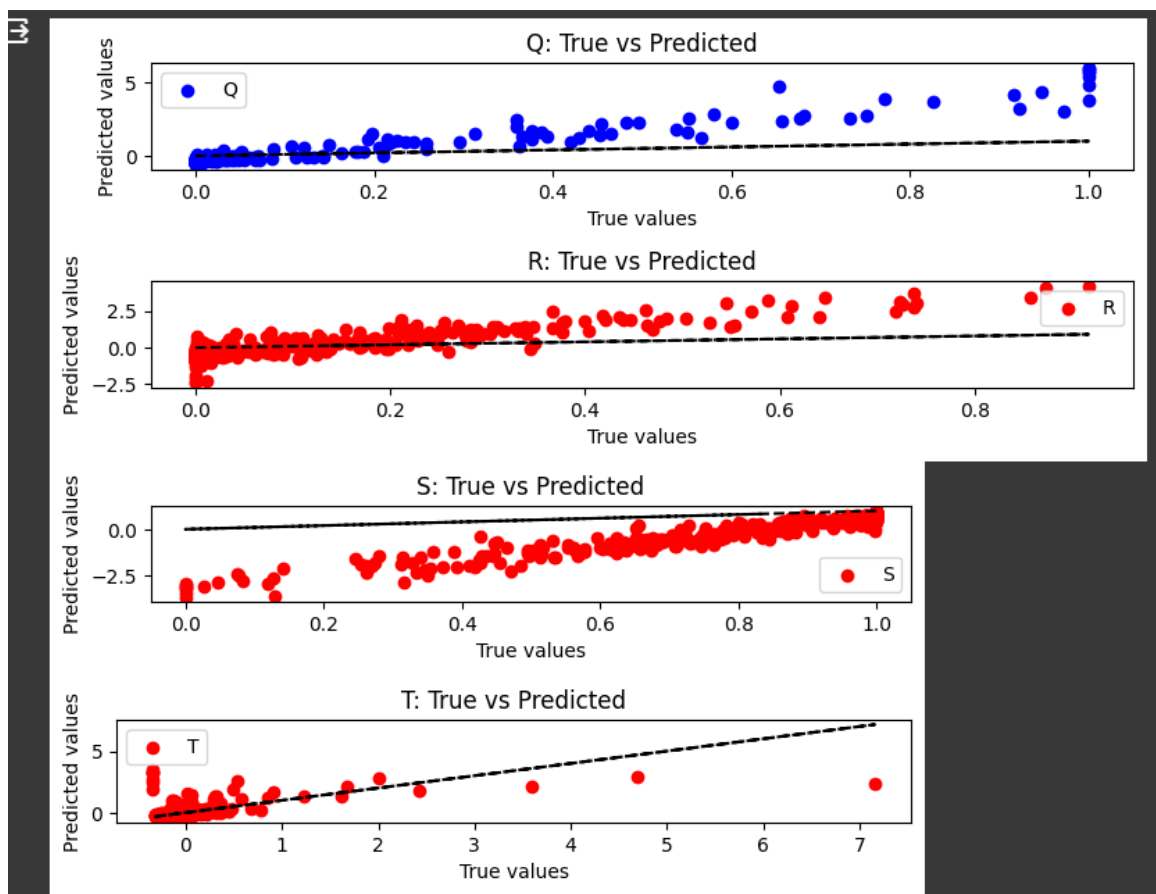
Sam model nie jest skomplikowany – składa się z 3 warstw gęstych i korzysta z funkcji aktywacji relu. 3 warstwy na taką ilość danych wydają się wystarczające.

```
[ ] criterion = nn.MSELoss()  
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-5)
```

Jako loss function wykorzystałem MSE, algorytm jako najszybszego spadku – Adam. Learning rate ustawiłem na 0.001 w myśl zasady największy dobrze działający oraz dodałem regularyzację L2, aby pilnować żeby żadna waga nie „wybuchła” i nie spowodowała słabszych osiągnięć modelu.



Po 1000 epok model przestał drastycznie się poprawiać, więc uznałem że na potrzeby testowania jest wystarczająco nauczony.



Tutaj zależy nam, aby wartości (kropki na wykresie) znajdowały się jak najbliżej narysowanej prostej.

```
[ ] model.eval()
    with torch.no_grad():
        predicted = model(features_test_s_m_t)

    r_squared_values = [r2_score(labels_test_s_m_t[:, i].numpy(), predicted[:, i].numpy()) for i in range(4)]

    print(r_squared_values)
    average_r_squared = np.mean(r_squared_values)

    print(f"R-squared: {average_r_squared}")

[0.9359459319254306, 0.8262476510883454, 0.9393703832653044, -0.12425061739748888]
R-squared: 0.6443283372203978
```

Do oceny modelu wykorzystałem wskaźnik błędu R-kwadrat (R-squared), który ocenia, jak dobrze model regresji dopasowuje się do danych. R-kwadrat jest liczony na podstawie odchylenia obserwacji od linii regresji, a jego wartość mieści się w przedziale od 0 do 1. Im bliżej wartości 1, tym lepiej model dopasowuje się do danych. W skrócie, R-kwadrat mierzy, jaka część zmienności zmiennych zależnych jest wyjaśniona przez model regresji. Oczywiście wysoki  $R^2$  nie zawsze oznacza, że model jest dobry.

Model dobrze radzi sobie z przewidywaniem pierwszych 3 wartości i kompletnie polega na ostatniej „T”.

```
[ ] prediction_data = [{'A': 7, 'B': 38.624, 'C': 24.35, 'D': 29.624, 'E': 0.624, 'F': 1.143, 'P': 0}]
```

```
model.eval()
with torch.no_grad():
    new_predictions = model(new_features_tensor)
    print(new_predictions)

pred = pd.DataFrame(new_predictions, columns=columns)

predicted_original_scale = (pred * std_devs_l) + means_l
print(predicted_original_scale)
```

```
tensor([[ -0.3671,  1.5923, -0.8180, -0.2945]])
      Q      R      S      T
0 -0.001143  0.371635  0.622845 1128.620302
```

Jeżeli chodzi o predykcje na nowym zestawie danych to wyniki wychodzą dość sensowne. Z jakiegoś powodu target „Q” zawsze jest ujemny i nie wiem jak to wytłumaczyć. (Opisany kod znajduje się w części pierwszej notatnika jupyterowskiego).

Aby poprawić wynik predykcji targetu „T” zdecydowałem się wprowadzeniu zmian do sposobu standaryzacji danych. Zamiast standaryzować cały target data, postanowiłem dokonać tej operacji tylko na samym targecie „T”. (Kod do tego rozwiązania znajduje się drugiej części notatnika).

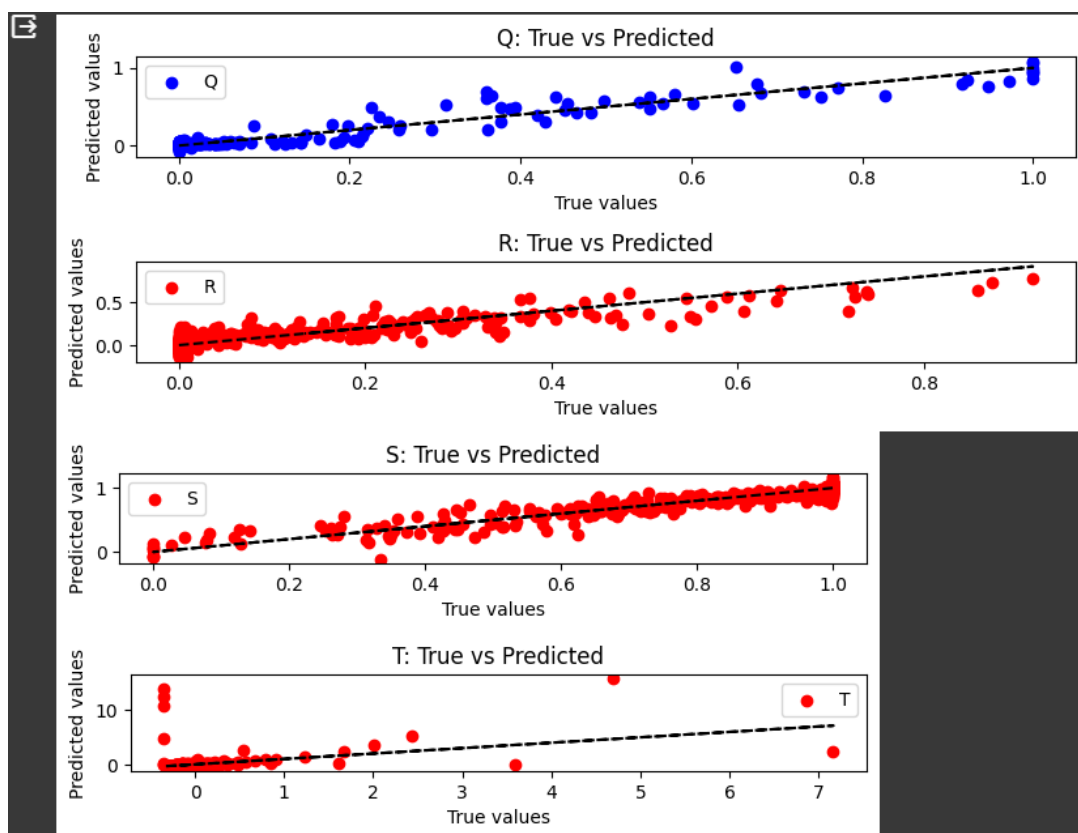
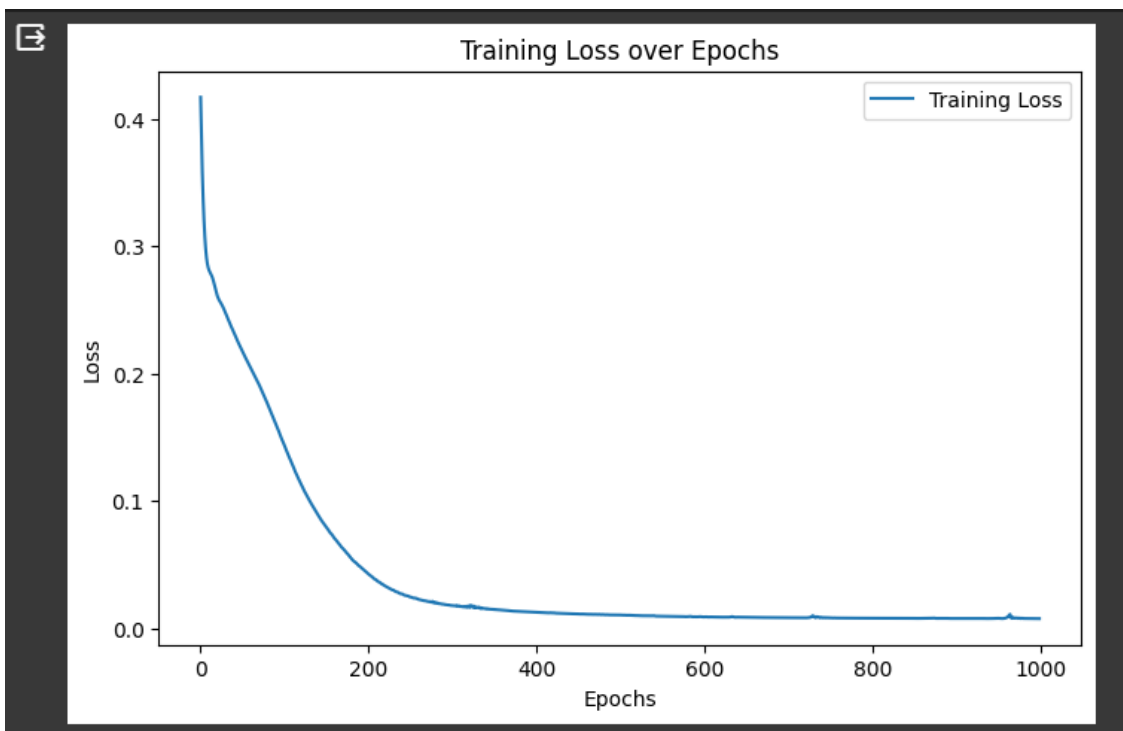
```
means_t = np.mean(labels_train["T"], axis=0)
std_devs_t = np.std(labels_train["T"], axis=0)

labels_train_s_m_t = (labels_train["T"] - means_t) / std_devs_t
labels_test_s_m_t = (labels_test["T"] - means_t) / std_devs_t

] features_train_s_m["P"] = features_train_s_m['P'].fillna(0)
  features_test_s_m["P"] = features_test_s_m['P'].fillna(0)

] labels_train["T"] = labels_train_s_m_t
  labels_test["T"] = labels_test_s_m_t
```

Wykorzystałem ten sam model z tymi samymi hiperparametrami oprócz learning rate, który zmieniłem na 0.01 nauczone doświadczeniem z poprzedniego przykładu, gdzie loss zaczął się stabilizować dopiero w obrębie tysięcznej epoki.





```
[ ] model.eval()
    with torch.no_grad():
        predicted = model(features_test_s_m_t)

    r_squared_values = [r2_score(labels_test[:, i].numpy(), predicted[:, i].numpy()) for i in range(4)]

    print(r_squared_values)
    average_r_squared = np.mean(r_squared_values)

    print(f"R-squared: {average_r_squared}")
```

[0.930395963622111, 0.786638058859918, 0.886313594685429, -4.763574398532462]

Model dobrze radzi sobie z 3 pierwszymi predykcjami, ale znowu ma problem z ostatnim targetem.

Na tym etapie doszedłem do wniosku, że próby standaryzacji nie przyniosły pożądanych skutków – wynika to z tego, że są one wrażliwe na obserwacje odstające (a takich w kolumnie „t” jest sporo) i dużą różnicę wartości pomiędzy najmniejszą i największą liczbą. Innych technik normalizacji nie ma co próbować, bo są jeszcze wrażliwsze na obserwacje odstające.

#### 4. 2 modele – jeden przewidujący „Q”, „R”, „S”, drugi – „T”

Przygotowanie danych do pierwszego modelu odbywa się w większości w ten sam sposób, nie standaryzuje jedynie targetów. Postanowiłem nie uwzględnić targetu „T” w uczeniu z racji na jego problematyczność.

features\_1=df[["A", "B", "C", "D", "E", "F", "P"]]  
features\_1

	A	B	C	D	E	F	P
0	7	37.281	22.982	27.478	0.535	1.007	0.0
1	4	44.585	24.704	37.649	0.488	1.301	0.0
2	6	44.580	28.450	28.923	0.774	0.908	0.0
3	5	45.122	18.719	41.147	1.280	1.378	0.0
4	3	34.953	19.789	25.372	0.917	0.605	0.0
...	...	...	...	...	...	...	...
2994	5	27.496	20.419	24.558	1.006	0.952	0.0
2995	4	36.921	16.461	29.046	0.410	1.226	0.0
2996	3	25.185	14.926	23.514	0.886	0.905	0.0
2997	3	41.186	22.007	32.554	0.702	1.344	0.0
2998	9	29.590	20.583	24.093	0.677	0.749	0.0

2999 rows x 7 columns

labels\_1 = df[["Q", "R", "S"]]  
labels\_1

	Q	R	S
0	0.006	0.251	0.743
1	0.000	0.001	0.999
2	0.060	0.288	0.652
3	0.000	0.000	1.000
4	0.000	0.000	1.000
...	...	...	...
2994	0.017	0.148	0.835
2995	0.000	0.000	1.000
2996	0.000	0.000	1.000
2997	0.000	0.000	1.000
2998	0.054	0.284	0.662

2999 rows x 3 columns

```

means = np.mean(features_train_1, axis=0)
std_devs = np.std(features_train_1, axis=0)

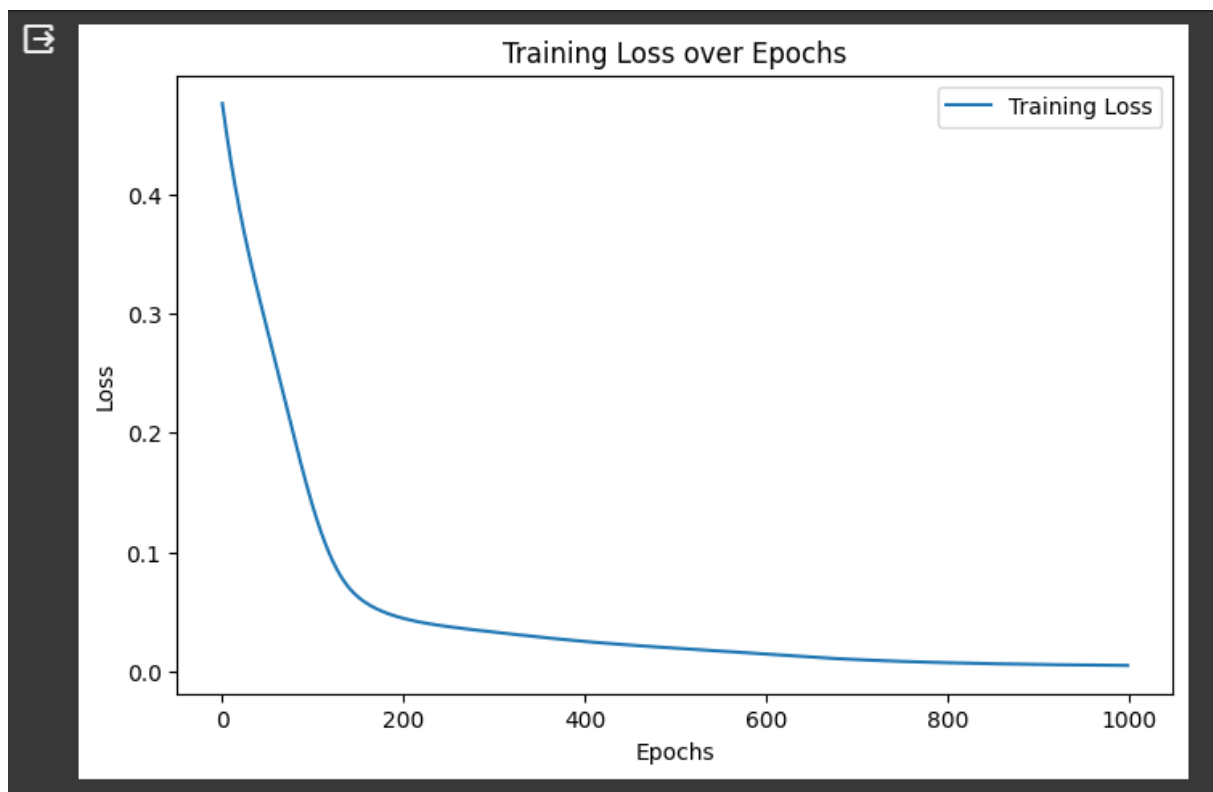
features_train_s_m = (features_train_1 - means) / std_devs
features_test_s_m = (features_test_1 - means) / std_devs

#means_l = np.mean(labels_train_1, axis=0)
#std_devs_l = np.std(labels_train_1, axis=0)

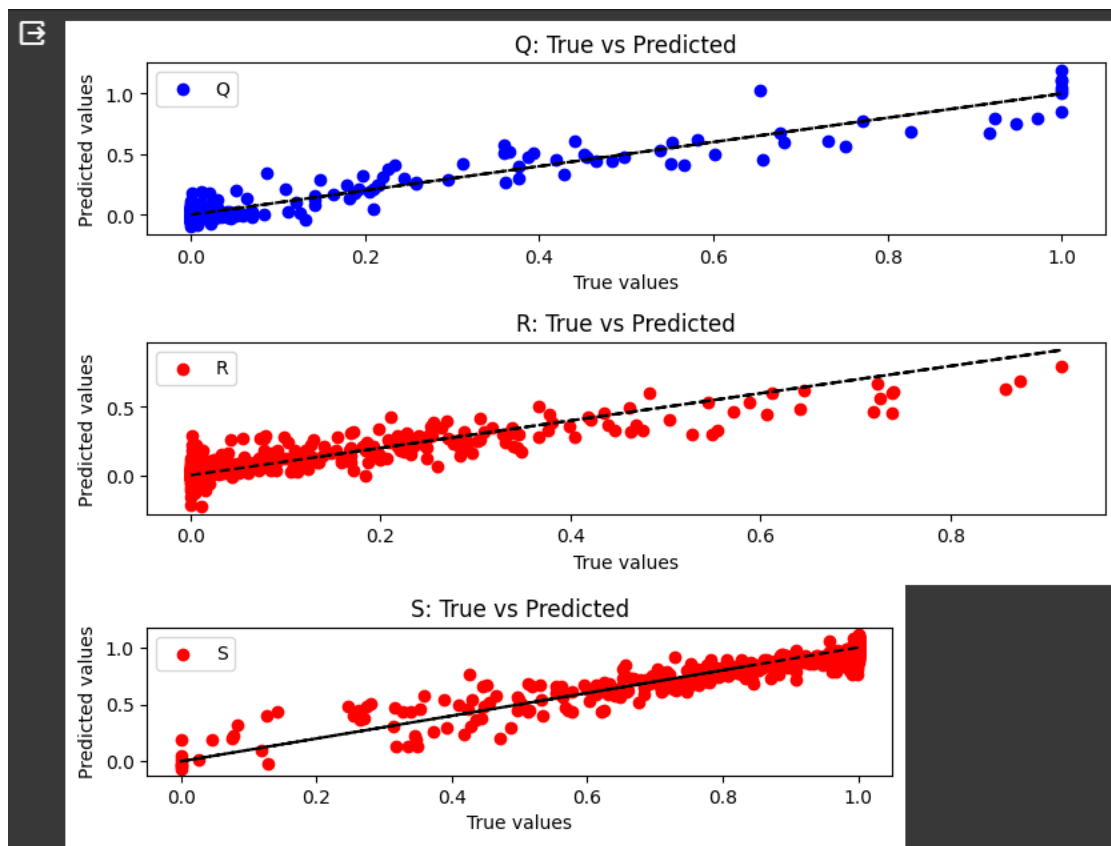
#labels_train_s_m = (labels_train_1 - means_l) / std_devs_l
#labels_test_s_m = (labels_test_1 - means_l) / std_devs_l

```

Ten model ma takie same hiperparametry i strukturę jak pierwszy model z części pierwszej notatnika.



Loss zaczął stabilizować się w granicach 200 epoki



```
[ ] model.eval()
    with torch.no_grad():
        predicted = model(features_test_s_m_t)

    r_squared_values = [r2_score(labels_test_s_m_t[:, i].numpy(), predicted[:, i].numpy()) for i in range(3)]

    print(r_squared_values)
    average_r_squared = np.mean(r_squared_values)

    print(f'R-squared: {average_r_squared}')
```

[0.9477046040469442, 0.861374981295497, 0.9262741514271244]  
R-squared: 0.9117845789231885

Wszystkie metryki wskazują, że model bardzo dobrze radzi sobie z powierzonym mu zadaniem.

```
[ ] prediction_data = [{'A': 7, 'B': 38.624, 'C': 24.35, 'D': 29.624, 'E': 0.624, 'F': 1.143, 'P': 0}]
df_predictions = pd.DataFrame(prediction_data)
df_predictions
```

```
model.eval()
with torch.no_grad():
    new_predictions = model(new_features_tensor)
    print(new_predictions)

pred = pd.DataFrame(new_predictions, columns=columns)

predicted_original_scale = (pred * std_devs_l) + means_l
print(predicted_original_scale)
```

```
tensor([[ -0.3698,  1.6093, -0.7487]])
      Q      R      S
0 -0.001683  0.374458  0.63979
```

Predykcje na nowych danych też wyglądają sensownie (stwierdziłem to na podstawie danych w liście prediction\_data, które specjalnie dobrałem, aby były podobne do jednej z obserwacji).

```
[ ] labels_2 = df[["T"]].# "Q", "R", "S", "T"
labels_2
```

	T
0	4190.292
1	6014.482
2	4200.343
3	19121.405
4	1978.472
...	...
2994	6489.224
2995	2450.382
2996	2723.639
2997	3792.123
2998	5151.800

2999 rows x 1 columns

```
[ ] features_2=df[["A", "B", "C", "D", "E", "F", "P", "Q", "R", "S"]]
features_2
```

	A	B	C	D	E	F	P	Q	R	S
0	7	37.281	22.982	27.478	0.535	1.007	0.0	0.006	0.251	0.743
1	4	44.585	24.704	37.649	0.488	1.301	0.0	0.000	0.001	0.999
2	6	44.580	28.450	28.923	0.774	0.908	0.0	0.060	0.288	0.652
3	5	45.122	18.719	41.147	1.280	1.378	0.0	0.000	0.000	1.000
4	3	34.953	19.789	25.372	0.917	0.605	0.0	0.000	0.000	1.000
...	...	...	...	...	...	...	...	...	...	...
2994	5	27.496	20.419	24.558	1.006	0.952	0.0	0.017	0.148	0.835
2995	4	36.921	16.461	29.046	0.410	1.226	0.0	0.000	0.000	1.000
2996	3	25.185	14.926	23.514	0.886	0.905	0.0	0.000	0.000	1.000
2997	3	41.186	22.007	32.554	0.702	1.344	0.0	0.000	0.000	1.000
2998	9	29.590	20.583	24.093	0.677	0.749	0.0	0.054	0.284	0.662

2999 rows x 10 columns

```

means = np.mean(features_train_2, axis=0)
std_devs = np.std(features_train_2, axis=0)

features_train_s_m = (features_train_2 - means) / std_devs
features_test_s_m = (features_test_2 - means) / std_devs

means_l = np.mean(labels_train_2, axis=0)
std_devs_l = np.std(labels_train_2, axis=0)

labels_train_s_m = (labels_train_2 - means_l) / std_devs_l
labels_test_s_m = (labels_test_2 - means_l) / std_devs_l

```

Tutaj ponownie byłem zmuszony standaryzować target, mimo wiedzy, że to nie do końca działa w moim przypadku.

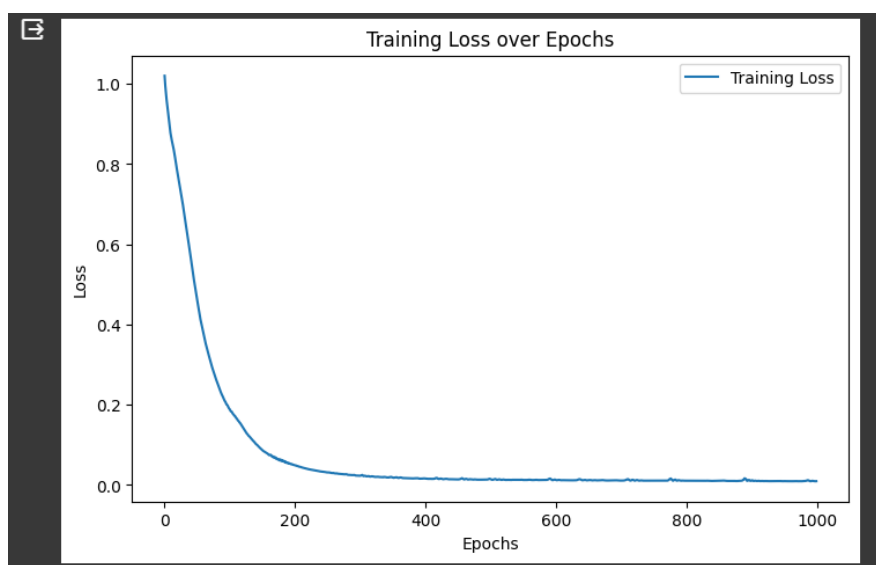
```

class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.fc1 = nn.Linear(7, 24)
        self.fc2 = nn.Linear(24, 10)
        self.fc3 = nn.Linear(10, 1)

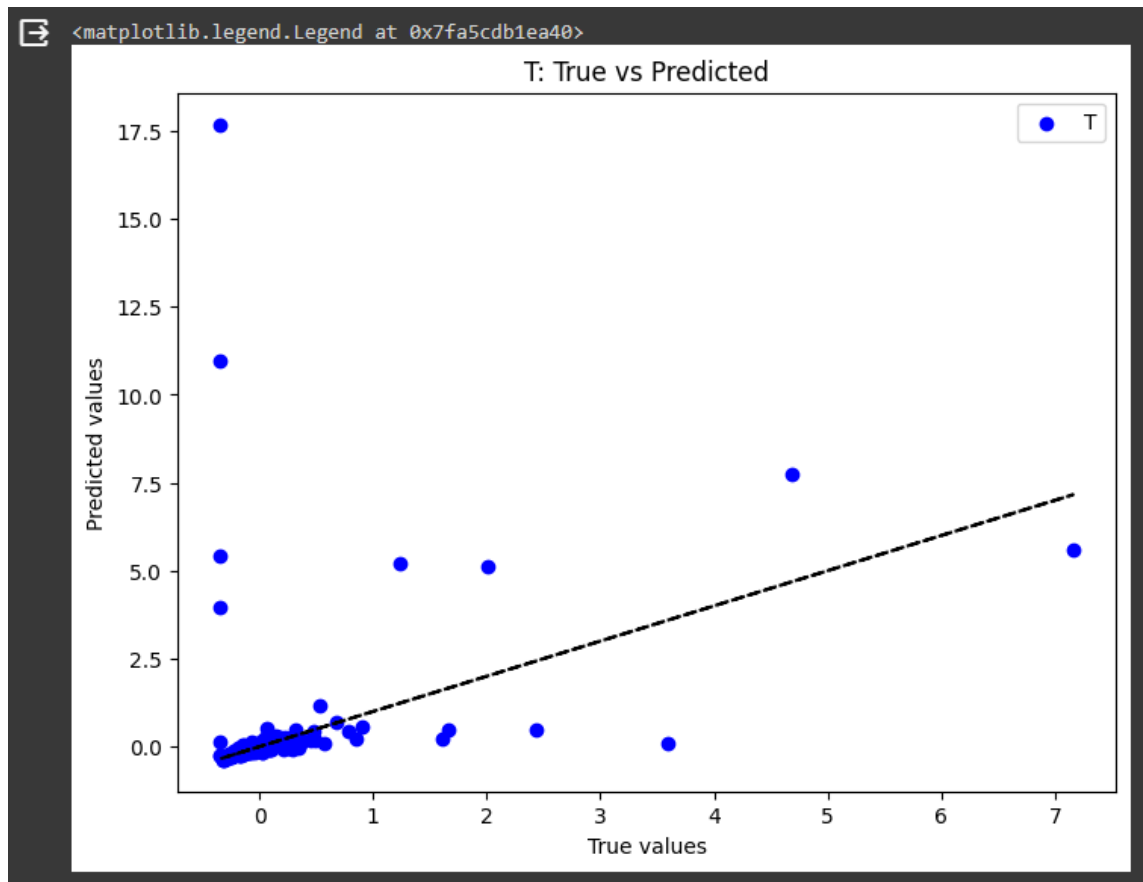
    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x

```

Sieć różni się od poprzednich tylko ilością neuronów. Parametry uczenia również są takie same jak w poprzedniej.



Model ustabilizował się na poziomie 200 epoki. Do 1000 epoki loss praktycznie nie spadał, a nawet momentami zwiększał się co mogło sugerować lekkie przeuczenie.



```
model.eval()
with torch.no_grad():
    predicted = model(features_test_s_m_t)

r_squared_values = [r2_score(labels_test_s_m_t[:, i].numpy(), predicted[:, i].numpy()) for i in range(1)]

print(r_squared_values)
average_r_squared = np.mean(r_squared_values)

print(f"R-squared: {average_r_squared}")
```

[ -3.658162923695233 ]  
R-squared: -3.658162923695233

Generalnie  $R^2$  jest bardzo słabe. Model radzi sobie trochę z przewidywaniem małych wartości, ale polega na większych.

Podsumowując - model, którego zadaniem było przewidywanie 4 wartości na raz z ustandaryzowanymi wszystkimi targetami poradził sobie najlepiej (choć stwierdzenie to nie świadczy o tym, że model jest bardzo dobry). Próby poradzenia sobie z targetem „T” używając standaryzacji nie przyniosły pożądaných skutków. Wynika to z tego, iż są one wrażliwe na obserwacje odstające (a takich w kolumnie „t” jest sporo) i dużą różnicę wartości pomiędzy najmniejszą i największą liczbą. Gdybym miał dalej rozwijać ten projekt to zdecydowałbym się na przeskalowanie danych w targetcie „t” do bardziej sensownych wartości i na pewno musiałbym znaleźć rozwiązanie problemu obserwacji odstających, aby standaryzacja miała większy sens i lepiej działała.