Grafowe Sieci Neuronowe

Michał Sumiński, Jan Płoszaj

1. Model klasyfikacji

Do trenowania modelu wykorzystana została funkcja CrossEntropyLoss, a optymalizator to Adam z parametrem learning_rate = 0.001. Aby zapobiegać przeuczeniu zastosowano metodę early stoppingu z parametrem patience = 10.

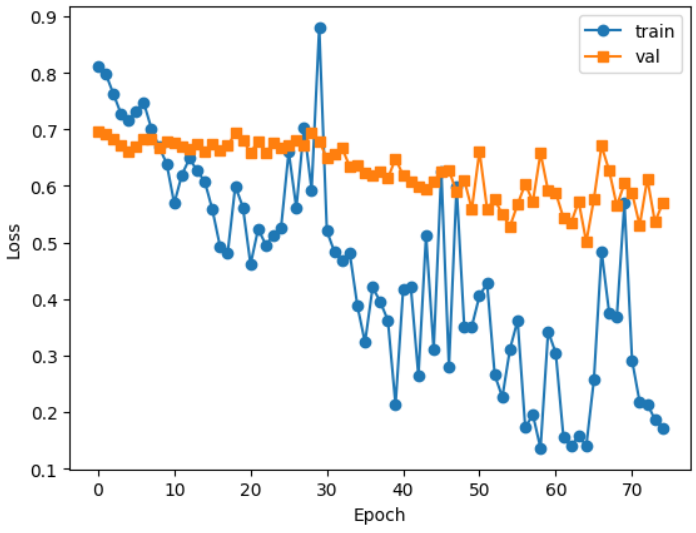
Badając warstwy TransfomerConv oraz GATConv z rozmiarami osadzeń 1 i 2 zauważyliśmy, że TransformerConv dawał wyraźnie lepsze wyniki w każdym przypadku, dlatego do poszukiwania optymalnego modelu również użyliśmy warstw grafowych TransformerConv.

Przetestowane kombinację modelu:

Liczba warstw grafowych	Rozmiar osadzenia	Predyktor	Accuracy
4	5	Nieliniowy (hidden_size = 4)	78%
4	10	Nieliniowy (hidden_size = 4)	77%
4	25	Nieliniowy (hidden_size = 4)	51%
4	50	Nieliniowy (hidden_size = 4)	51%
4	5	Liniowy	77%
4	10	Liniowy	76%
3	5	Nieliniowy (hidden_size = 4)	77%
5	5	Nieliniowy (hidden_size = 4)	51%

Zatem optymalny model wyglądał następująco:

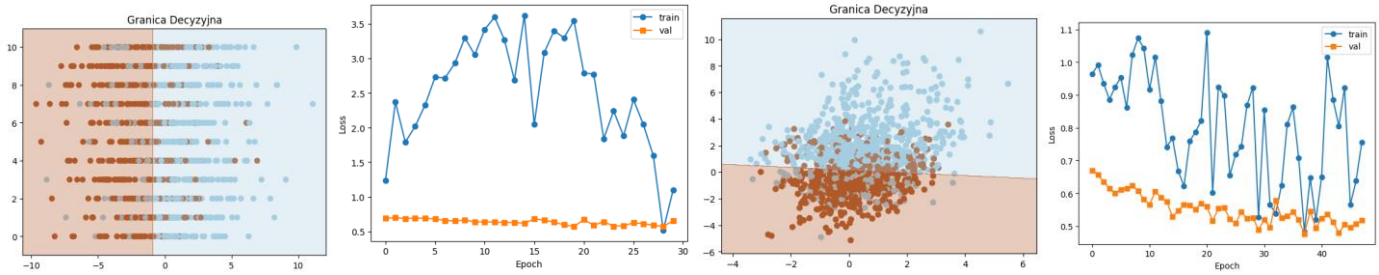
```python
class CustomTransformerModelPerceptron(nn.Module):
    def __init__(self, out_channels, hidden_layer_size):
        super(CustomTransformerModelPerceptron, self).__init__()

        edge_dim=3 # edge_attr.size(1)

        self.conv1 = TransformerConv(dataset.num_features, 32, edge_dim=edge_dim)
        self.conv2 = TransformerConv(32, 64, edge_dim=edge_dim)
        self.conv3 = TransformerConv(64, 64, edge_dim=edge_dim)
        self.conv4 = TransformerConv(64, out_channels, edge_dim=edge_dim)

        self.relu = nn.ReLU()

        self.linear1 = nn.Linear(out_channels, hidden_layer_size)
        self.linear2 = nn.Linear(hidden_layer_size, 2) # klasyfikacja binarna

    def forward(self, data, flag):
        if flag:
            x = data.x.to(dtype=torch.float32)
            edge_index = data.edge_index
            edge_attr = data.edge_attr.to(dtype=torch.float32)

            x = self.conv1(x, edge_index, edge_attr)
            x = self.relu(x)

            x = self.conv2(x, edge_index, edge_attr)
            x = self.relu(x)

            x = self.conv3(x, edge_index, edge_attr)
            x = self.relu(x)

            x = self.conv4(x, edge_index, edge_attr)

            feature = torch.mean(x, dim=0)

            x = self.linear1(feature)

            x = self.relu(x)

            x = self.linear2(x)

            return x, feature
```



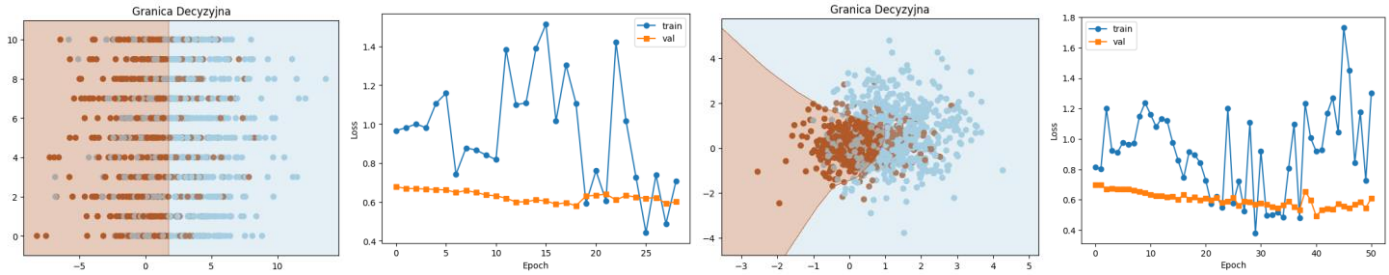
Wyniki testów dla osadzeń 1 i 2 oraz ich wizualizacje:

*UWAGA: osadzenia jednoelementowe zostały przedstawione w 2D w celu lepszej czytelności wyników, współrzędna y jest wygenerowaną losową liczbą z przedziału [0,10], oczywiście wynikiem tego osadzenia jest tylko wartość współrzędnej x

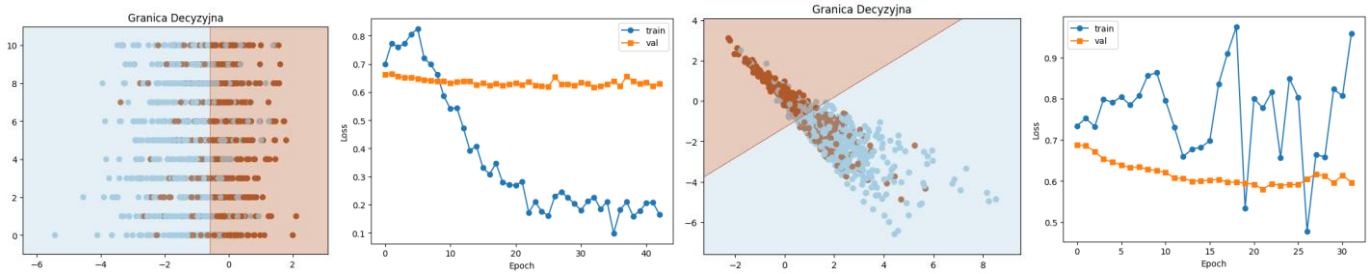
TransformerConv osadzenie = 1 , Test_accuracy = 0.70, predyktor liniowy (po lewej) oraz TransformerConv osadzenie = 2 , Test_accuracy = 0.72, predyktor liniowy (po prawej)



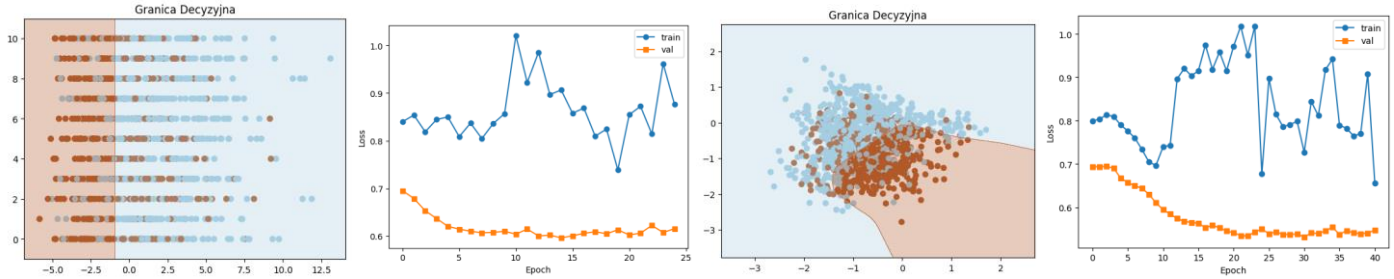
TransformerConv osadzenie = 1 , Test_accuracy = 0.74, perceptron ukryta = 128 (po lewej) TransformerConv osadzenie = 2 , Test_accuracy = 0.74, perceptron ukryta = 128 (po prawej)



GATConv osadzenie = 1 , Test_accuracy = 0.69, predyktor liniowy (po lewej) oraz GATConv osadzenie = 2 , Test_accuracy = 0.70, predyktor liniowy (po prawej)



GATConv osadzenie = 1 , Test_accuracy = 0.625, perceptron ukryta = 4 (po lewej) oraz GATConv osadzenie = 2 , Test_accuracy = 0.68, perceptron ukryta = 128 (po prawej)



Funkcje aproksymująca (regresja) do przewidywania momentu dipolowego dla cząstek ze zbioru QM9.

LinearModel (predyktor liniowy)

```python
class LinearModel(torch.nn.Module):
    def __init__(self, out_channel, dim_h):
        super().__init__()
        self.conv1 = GCNConv(qm9.num_features, dim_h)
        self.conv2 = GCNConv(dim_h, dim_h)
        self.conv3 = GCNConv(dim_h, out_channel)
        self.lin = torch.nn.Linear(out_channel, 1)

    def forward(self, data, flag):
        if flag:
          e = data.edge_index
          x = data.x

          x = self.conv1(x, e)
          x = x.relu()
          x = self.conv2(x, e)
          x = x.relu()
          x = self.conv3(x, e)
          feature = global_mean_pool(x, data.batch)
          x = Fun.dropout(feature, p=0.5, training=self.training)
          x = self.lin(x)
          return x, feature
```

PerceptronModel (predyktor nieliniowy)

```python
class PerceptronModel(torch.nn.Module):
    def __init__(self, out_channel, dim_h):
        super().__init__()
        self.conv1 = GCNConv(qm9.num_features, dim_h)
        self.conv2 = GCNConv(dim_h, dim_h)
        self.conv3 = GCNConv(dim_h, out_channel)

        self.linear1 = nn.Linear(out_channel, dim_h)
        self.linear2 = nn.Linear(dim_h, 1)

    def forward(self, data, flag):
      if flag:
        e = data.edge_index
        x = data.x

        x = self.conv1(x, e)
        x = x.relu()
        x = self.conv2(x, e)
        x = x.relu()
        x = self.conv3(x, e)
        features = global_mean_pool(x, data.batch)

        x = Fun.dropout(features, p=0.5, training=self.training)
        x = self.linear1(x)
        x = x.relu()
        x = self.linear2(x)
        return x, features
```

Warunkiem stopu w uczeniu modeli była liczba epok, jednak, jeśli loss dla zbioru walidacyjnego w trakcie nauki był mniejszy niż globalnie najmniejszy do tej pory, to zapisywano model do pliku. Dzięki temu po osiągnięciu wymaganej liczby epok osiągano optymalny model. Do optymalizacji modelu wykorzystano algorytm Adam (torch.optim.Adam). Współczynnik uczenia (lr) został ustawiony na 0.001. Jako funkcję straty wybrano średni błąd kwadratowy (MSELoss). Jakość modelu była oceniania z użyciem miary średniego błędu bezwzględnego (MAE).

Proces wyboru optymalnej architektury dla osadzenia 1 (kolumna 1, 2) / dla osadzenia 2 (kolumna 3, 4)

- testy liczby warstw grafowych dla 30 epok, dropout=0.5, liczby warstw ukrytych = 4
    - predyktor liniowy

Liczba warstw grafowych GCNConv	MAE	Liczba warstw grafowych GCNConv	MAE
2	0.723	2	0.687
3	0.704	3	0.682

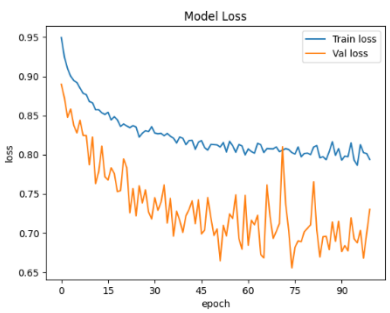
    - predyktor nieliniowy

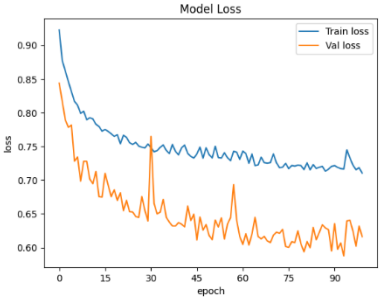
Liczba warstw grafowych GCNConv	MAE	Liczba warstw grafowych GCNConv	MAE
2	0.889	2	0.664
3	0.653	3	0.635

- testy liczby warstw ukrytych dla 30 epok, dropout=0.5, liczba warstw grafowych = 3, liczba osadzeń = 1, predyktor liniowy:

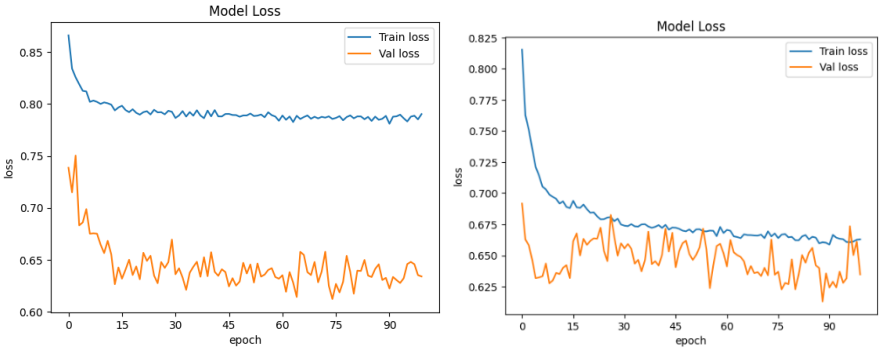
Liczba warstw ukrytych	MAE
4	0.704
16	0.693
64	0.651

Proces uczenia optymalnych architektur: LinearModel(100 epok, dropout=0.5, liczby warstw ukrytych = 64, 3 warstwy grafowe GCNConv, liczba osadzeń =1 (1 kolumna) i 2 (2 kolumna), predyktor liniowy





PerceptronModel (100 epok, dropout=0.5, liczby warstw ukrytych = 64, 3 warstwy grafowe GCNConv, liczba osadzeń = 1 (1 kolumna) i 2 (2 kolumna), predyktor nieliniowy

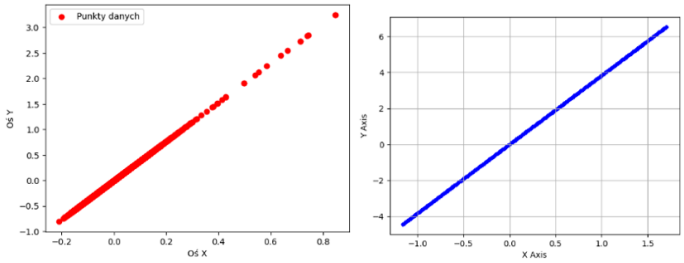


Wyniki testów dla osadzeń 1 i 2 oraz ich wizualizacje:

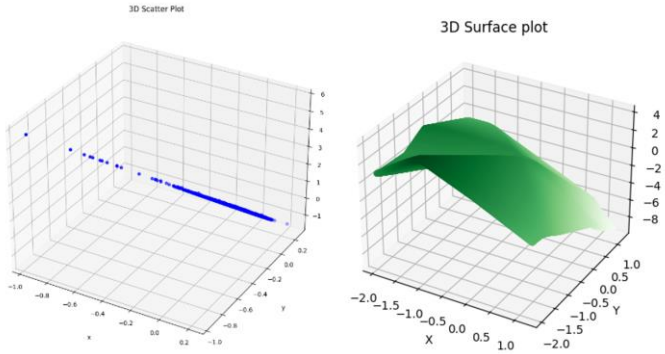
Zaprezentowano pary wykresów: predykcja dla zbioru testowego i osiągnięta funkcja aproksymująca:

Predyktor liniowy

Osadzenie 1, MAE = 0.654



Osadzenie 2, MAE = 0.596



Osadzenie 2, MAE=0.606

Predyktor nieliniowy

Osadzenie 1, MAE = 0.592