Lab02
Sieci Neuronowe

Wynikiem implementacji listy jest program podzielony na 3 pliki:

- Functions.py – funkcje pomocnicze, dyskretyzacja danych, normalizacja danych, obliczenie metryk dla zestawu testowego.
- Main.py – środowisko testowe, w nim testuje wizualizuje zaimplementowane metody i funkcje.
- NeuralNetwork.py – klasa prostej sieci neuronowej, opartej na gradiencie z entropii krzyżowej.

Implementacja rozwiązania:

Wyjście w sieci było implementowane na wzór:

$$p(x) = \sigma(Wx + b)$$

```
def p(self, x):
    argument = np.dot(x, self.W) + self.b
    return self.sigmoid(argument)
```

gdzie funkcja sigmoid to:

$$\sigma(n) = \frac{1}{1 + e^{-n}}$$

Co w pythonie może być osiągnięte za pomocą funkcji [expit(x)] :

```
def sigmoid(self, n):
    return expit(n)
```

Jako funkcję kosztu wykorzystujemy entropię krzyżową:

$$L = -y \ln p(x) - (1 - y) \ln(1 - p(x))$$

```
def cross_entropy_loss(self, y, y_pred):
    epsilon = 1e-15
    loss = y * np.log(y_pred + epsilon) + (1 - y) * np.log(1 - y_pred + epsilon)
    return -np.sum(loss)
```

epsilon został dodany ponieważ był problem z liczeniem wartości 0.

Implementacja gradientu będzie za pomocą pochodnej po wagach modelu, co z entropii krzyżowej daje:

$$\frac{\partial L}{\partial w_i} = -(y - p(x))x_i$$

$$\frac{\partial L}{\partial w_i} = (p(x) - y)x_i$$

```
def compute_gradient(self, X_train, y_train):
    y_pred = self.p(X_train)
    dz = y_pred - y_train
    dw = np.dot(X_train.T, dz)
    db = np.sum(dz)
    return dw, db
```

Model uczy się na podstawie zmiany wag, tak aby iść w stronę wyznaczoną przez gradient. Implementacja:

$$w_i' = w_i - \alpha \frac{\partial L}{\partial \mathrm{wi}}$$

```
dw, db = self.compute_gradient(X_train, y_train)

# Aktualizacja wag i bias zgodnie z gradientem i współczynnikiem uczenia
self.W -= self.learning_rate * dw
self.b -= self.learning_rate * db
```

Mój model posiada również 3 funkcje które mogą go wyuczyć:

- Fit_model_covergence – który mówi o wystarczająco małej zmianie aby przerwać proces uczenia.
- Fit – podstawowa wersja ucząca model, przechodząca przez cały zbiór X razy.
- Fit_batches – wersja rozszerzona o dzielenie zbioru na paczki o nadanej wielkości, przechodzi przez zbiór X razy.

Aby zmaksymalizować uczenie się modelu, dane przed każdą iteracją są losowo mieszane.

Aby zobaczyć wpływ procesowania danych będę rozpatrywał wszystkie wyniki kontekście 3 metod procesowania:

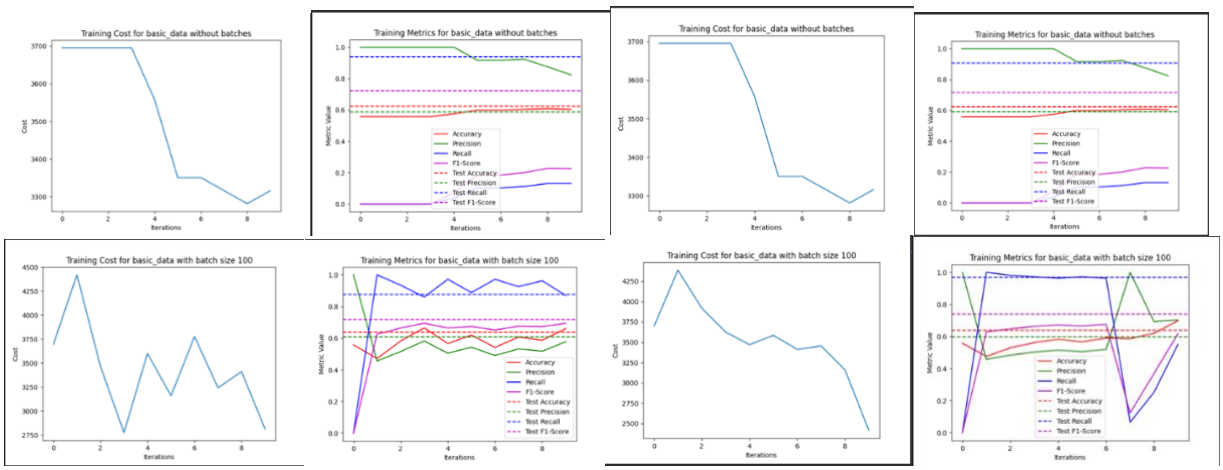
- Normalizacja
- Dyskretyzacja
- Surowe dane

Hiperparametry i parametry zostały wybrane dla każdego osobno tak aby zmaksymalizować ich potencjał.

Wyniki uczenia dla parametrow i hiperparametrów:

Surowe dane

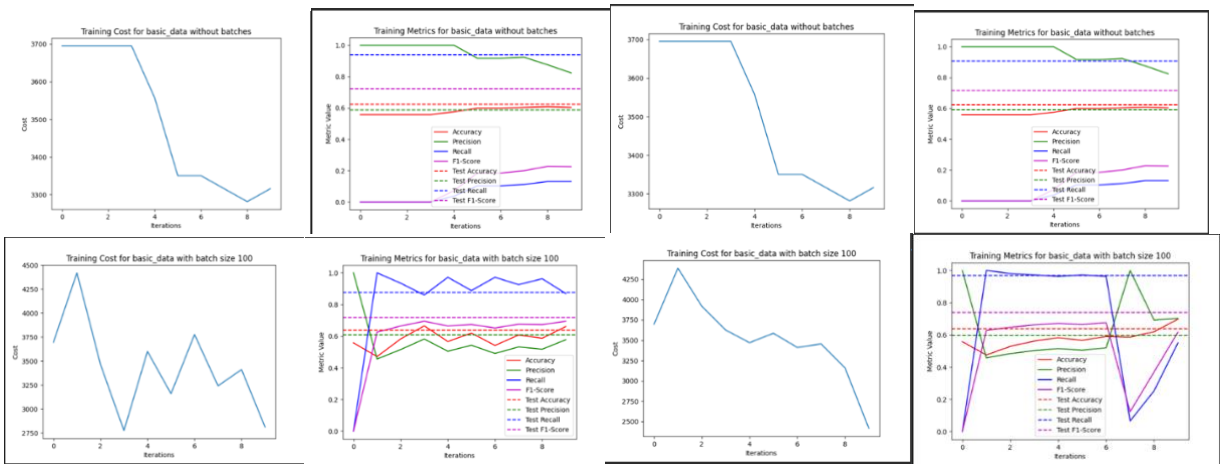
```
learning_rate_basic_without_b = 0.1
learning_rate_basic_with_b = 0.001
num_of_iterations_basic = 400
batch_size = 100
```



Możemy stąd zauważyć, że dane paczkowane, mają lepszy wynik ale są mniej stabilne jeśli chodzi o metryki i proces uczenia.
Jendak oba wyniki są bardzo dobre, plasują się na poziomie >=0.6 jeśli chodzi o wszystkie metryki, co jest lepsze niż losowe zgadywanie. Najlepsza

Surowe dane

```
learning_rate_basic_without_b = 0.1
learning_rate_basic_with_b = 0.001
num_of_iterations_basic = 400
batch_size = 100
```

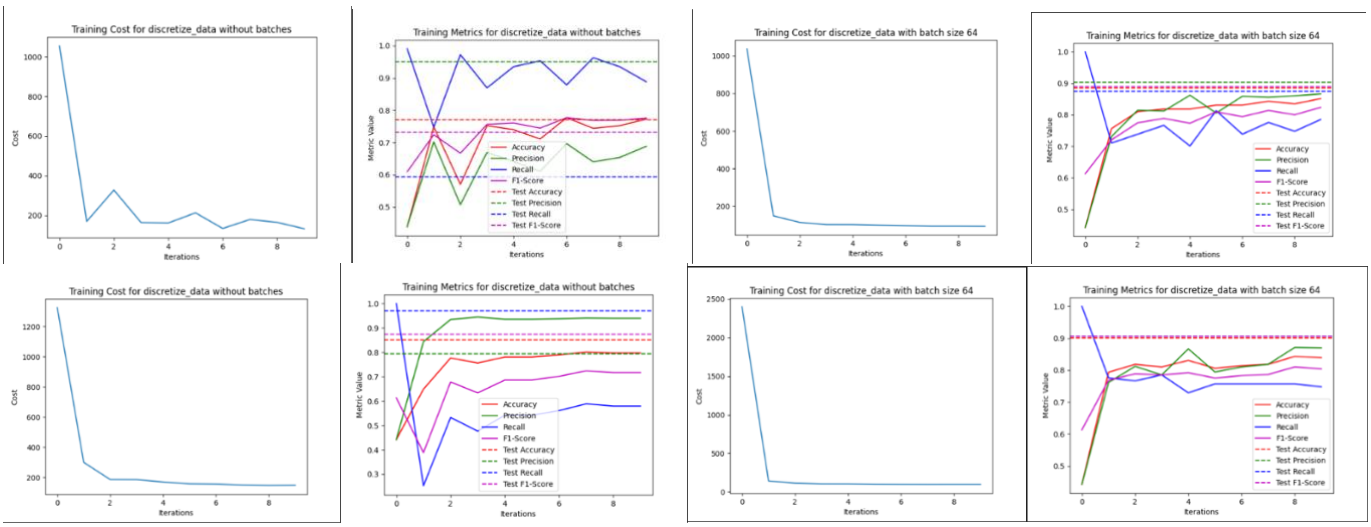


Możemy stąd zauważyć, że dane paczkowane, mają lepszy wynik ale są mniej stabilne jeśli chodzi o metryki i proces uczenia.

Najlepiej wypada tutaj czułość która mówi, ile z faktycznych pozytywnych przypadków jest identyfikowanych. Widzimy że jest to ok. 90% co jest bardzo dobrym wynikiem.

Dane poddane dyskretyzacji:

```
learning_rate_discrete_without_b = 0.0005
learning_rate_discrete_with_b = 0.0005
batch_size = 64
num_of_iterations_discretization = 60
```

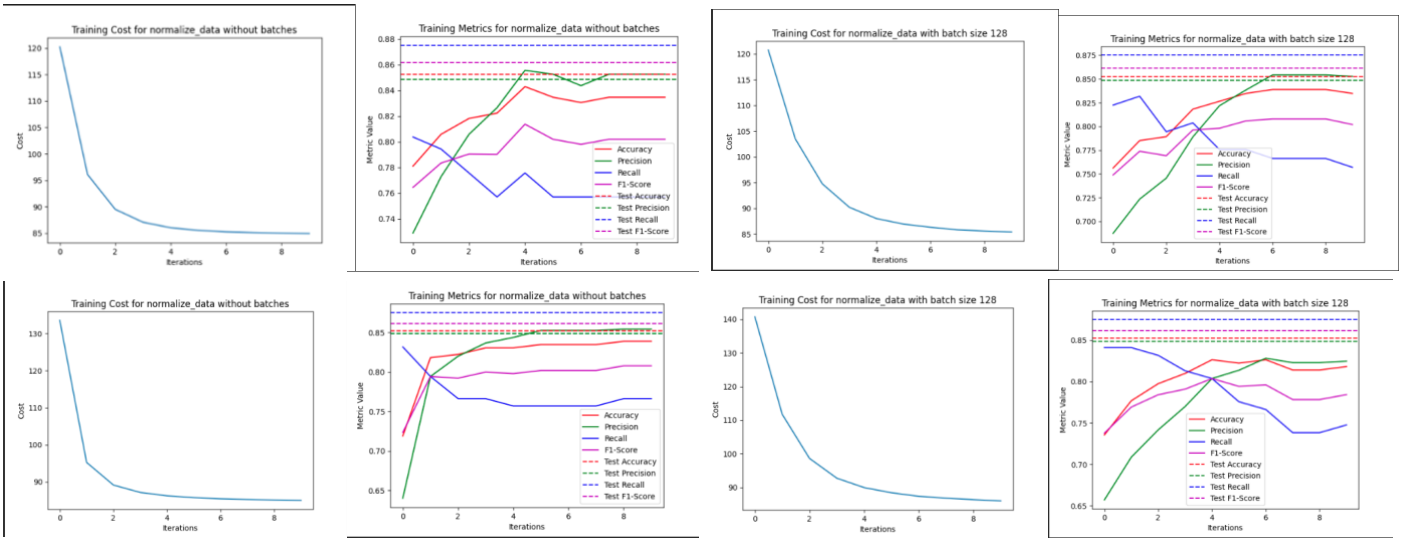


W tym przypadku możemy zauważyć, że jest zdecydowanie mniej iteracji bo tylko 60 i model się stabilizuje, w przypadku paczkowania pomimo braku wyraźnej różnicy na wykresie kosztu możemy zauważyć różnicę w miarach. Wszystkie testowe miary wskazują ok. 90% poprawności, co jest wspaniałym wynikiem.

W przypadku paczkowania możemy także zauważyć mniejsze wahania metryk.

Dane poddane normalizacji:

```
learning_rate_normalization_without_b = 0.001
learning_rate_normalization_with_b = 0.001
num_of_iterations_normalization = 200
batch_size= 128
```



W tym przypadku możemy zauważyć, że krzywa ucząca wygląda wzorowo. Metryki także to pokazują, są stabilne i utrzymują się na jednym poziomie. Przetestowany model ma skuteczność wszystkich metryk na poziomie >=85%. Normalizacja danych sprawdza się tutaj bardzo dobrze, może to być spowodowane rozkładem wartości większości atrybutów, który jest podobny do normalnego.

Wnioski:

Dobranie odpowiednich parametrów i hiperparametrów odgrywa kluczową rolę w powodzeniu modelu. Jest to ciężkie bez znajomości metod na znalezienie optmalnych współczynników, trzeba sprawdzać to metodą prób i błędów.

Preprocessing też odgrywa ważną rolę w tym jak sprawuje się model.

Nawet taki prosty model może sobie dobrze radzić z binarną klasyfikacją.