Politechnika Krakowska
Wydział Inżynierii
Elektrycznej i Komputerowej

Sztuczna Inteligencja

Projekt

„Zastosowanie sieci feedforward w wybranym problemie klasyfikacyjnym”

Mikołaj Knap, Kamil Rojek

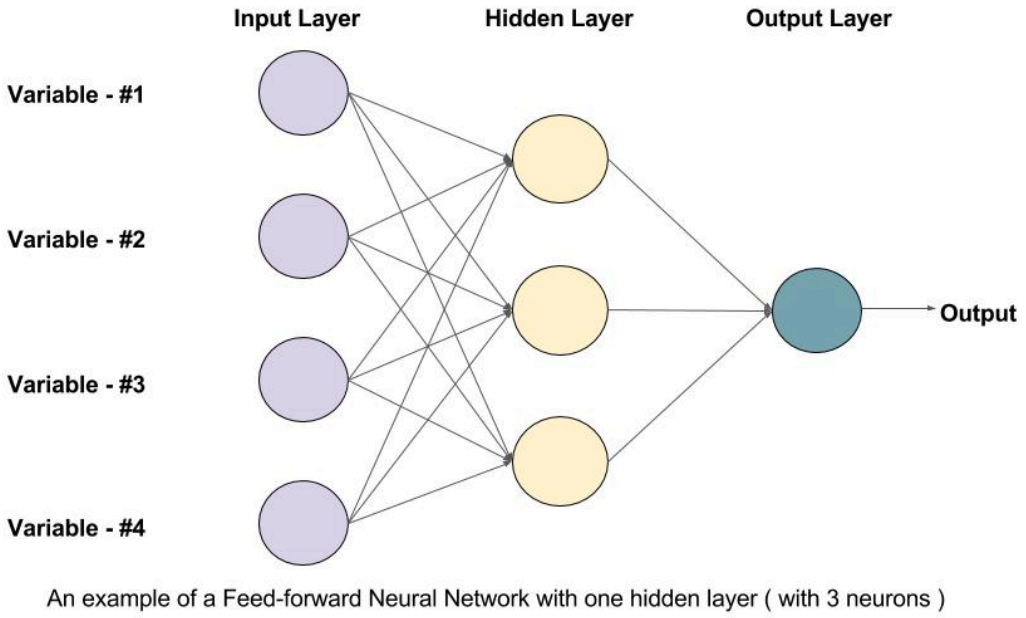
Informatyka w Inżynierii Komputerowej

Semestr 6

1. Wstęp teoretyczny

Sieć feedforward

Sieć feedforward to model sztucznej sieci neuronowej, w którym połączenia pomiędzy neuronami nie tworzą cykli, a dane przesyłane są tylko w jednym kierunku, od warstwy wejściowej do warstwy wyjściowej. Oznacza to, że dane przetwarzane są sekwencyjnie, bez cofania się do poprzednich warstw. Możemy w niej zdefiniować trzy warstwy: warstwa wejściowa, jedna lub więcej warstw ukrytych oraz warstwa wyjściowa. Każda warstwa składa się z neuronów, które odbierają dane wejściowe, przetwarzają je i przekazują do kolejnej warstwy.



Rysunek 1. Schemat sieci neuronowej feed-forward z jedną warstwą ukrytą z 3 neuronami

Klasyfikacja

Klasyfikacja to nadzorowana metoda uczenia maszynowego, w której model próbuje przewidzieć poprawną etykietę danych wejściowych. Podczas klasyfikacji model jest w pełni trenowany przy użyciu danych uczących, a następnie jest oceniany na danych testowych, zanim zostanie wykorzystany do przewidywania nowych.

W naszym projekcie zastosowany model jest multiklasyfikatorem, czyli realizuje zadanie klasyfikacji, w którym obiekty są przypisywane do jednej z wielu predefiniowanych klas.

Backpropagation

Propagacja wsteczna to algorytm uczenia maszynowego wykorzystywany do dostosowania wag połączeń między neuronami w sieci neuronowej w celu minimalizacji błędu prognozowania na danych treningowych.

W pierwszej fazie, nazywanej fazą propagacji, sygnał danych jest przekazywany od warstwy wejściowej do warstwy wyjściowej, a następnie obliczane są błędy wyjściowe. W drugiej fazie, nazywanej fazą wstecznej propagacji, błędy te są propagowane wstecz przez sieć, aby obliczyć błędy wewnątrz sieci i dostosować wagi połączeń, minimalizując tym samym błąd prognozowania.

CrossEntropyLoss

CrossEntropyLoss jest funkcją straty wykorzystywaną w zadaniach klasyfikacji, szczególnie w przypadku, gdy dane mają więcej niż dwie klasy. Jest ona często używana wraz z funkcją aktywacji softmax na ostatniej warstwie sieci neuronowej.

Z-score

Z-score to miara, która określa, jak daleko odchyla się dana wartość od średniej wartości punktu danych, mierzona w odchyleniach standardowych. Jest to znormalizowana wartość, która pozwala porównywać i interpretować pozycję danego punktu danych w kontekście rozkładu danych.

Wzór:

$$Z = \frac{(X - \mu)}{\sigma}$$

Gdzie:

X - wartość danych,
μ - średnia danych,
σ - odchylenie standardowe danych.

Wzór 1. Wzór na obliczanie Z-score

2. Opis danych wejściowych

1) Przedstawienie danych

Dane obejmują zestaw cech związanych z różnymi parametrami technicznymi telefonów komórkowych. Każda cecha ma swoje znaczenie i może mieć wpływ na cenę telefonu oraz jego wydajność.

Źródło danych: www.kaggle.com/datasets/atefehmirnaseri/cell-phone-price

2) Omówienie cech danych

- **battery_power**: Całkowita pojemność baterii, która może być przechowywana (mAh)
- **blue**: Czy urządzenie ma Bluetooth, tak/nie (1/0)
- **clock_speed**: Prędkość wykonywania instrukcji przez mikroprocesor
- **dual_sim**: Czy urządzenie obsługuje dwie karty SIM jednocześnie, tak/nie (1/0)
- **fc**: Jakość aparatu przedniego w Megapikselach
- **four_g**: Czy urządzenie obsługuje sieć 4G, tak/nie (1/0)
- **int_memory**: Pamięć wewnętrzna w gigabajtach
- **m_dep**: Grubość urządzenia w centymetrach
- **mobile_wt**: Waga urządzenia
- **n_cores**: Liczba rdzeni procesora
- **pc**: Jakość aparatu głównego w Megapikselach
- **px_height**: Wysokość rozdzielczości pikseli
- **px_width**: Szerokość rozdzielczości pikseli
- **ram**: Pamięć RAM w megabajtach
- **sc_h**: Wysokość ekranu urządzenia w centymetrach
- **sc_w**: Szerokość ekranu urządzenia w centymetrach
- **talk_time**: Maksymalny czas rozmowy, który może obsłużyć pełny naładowany akumulator urządzenia
- **three_g**: Czy urządzenie obsługuje sieć 3G, tak/nie (1/0)
- **touch_screen**: Czy urządzenie ma ekran dotykowy, tak/nie (1/0)
- **wifi**: Czy urządzenie obsługuje Wi-Fi, tak/nie (1/0)
- **price_range**: Zaklasyfikowany zakres cenowy urządzenia

3) Podstawowe informacje

Przedstawienie pierwszych 5 wierszy danych:

Dane treningowe:

	battery_power	blue	clock_speed	dual_sim	fc	four_g	int_memory	m_dep	mobile_wt	n_cores	...	px_height	px_width	ram	sc_h	sc_w	talk_time	three_g	touch_screen	wifi	price_range
0	842	0	2.2	0	1	0	7	0.6	188	2	...	20	756	2549	9	7	19	0	0	1	1
1	1021	1	0.5	1	0	1	53	0.7	136	3	...	905	1988	2631	17	3	7	1	1	0	2
2	563	1	0.5	1	2	1	41	0.9	145	5	...	1263	1716	2603	11	2	9	1	1	0	2
3	615	1	2.5	0	0	0	10	0.8	131	6	...	1216	1786	2769	16	8	11	1	0	0	2
4	1821	1	1.2	0	13	1	44	0.6	141	2	...	1208	1212	1411	8	2	15	1	1	0	1

Rysunek 2. Pierwsze 5 wierszy bazy danych

Wymiary danych:

Liczba wierszy: 2000
Liczba kolumn: 21

4) Analiza danych

Pierwsze 5 wierszy danych dostarcza wielu informacji o poszczególnych kolumnach. Widać, że niektóre kolumny są ciągłe, a niektóre są dyskretne. Poniżej zrobimy podział na kolumny ciągłe i dyskretne, a następnie za pomocą funkcji description na dataframe przeanalizujemy podstawowe informacje o zbiorze.

Kolumny ciągłe:

battery_power, clock_speed, fc, int_memory, m_dep, mobile_wt, pc, px_height, px_width, ram, sc_h, sc_w, talk_time

Kolumny dyskretne:

blue, dual_sim, four_g, n_cores, three_g, touch_screen, wifi, price_range

```
# Podział danych na kolumny ciągłe i dyskretne

df_train_c = df_train.loc[:, ['battery_power', 'clock_speed', 'fc', 'int_memory', 'm_dep', 'mobile_wt', 'pc', 'px_height', 'px_width', 'ram', 'sc_h', 'sc_w', 'talk_time']]
df_train_d = df_train.loc[:, ['blue', 'dual_sim', 'four_g', 'n_cores', 'three_g', 'touch_screen', 'wifi', 'price_range']]
```

Rysunek 3. Podział kolumn na ciągłe i dyskretne

```
# Dane treningowe
print("Opis danych treningowych:")
print("Kolumny ciągłe: ")
display(df_train_c.describe())
print("Kolumny dyskretne: ")
display(df_train_d.describe())
```

Rysunek 4. Program do wyświetlenia opisu kolumn ciągłych i dyskretnych

Opis danych treningowych:
Kolumny ciągłe:

	battery_power	clock_speed	fc	int_memory	m_dep	mobile_wt	pc	px_height	px_width	ram	sc_h	sc_w	talk_time
count	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000
mean	1238.518500	1.522250	4.309500	32.046500	0.501750	140.249000	9.916500	645.108000	1251.515500	2124.213000	12.306500	5.767000	11.011000
std	439.418206	0.816004	4.341444	18.145715	0.288416	35.399655	6.064315	443.780811	432.199447	1084.732044	4.213245	4.356398	5.463955
min	501.000000	0.500000	0.000000	2.000000	0.100000	80.000000	0.000000	0.000000	500.000000	256.000000	5.000000	0.000000	2.000000
25%	851.750000	0.700000	1.000000	16.000000	0.200000	109.000000	5.000000	282.750000	874.750000	1207.500000	9.000000	2.000000	6.000000
50%	1226.000000	1.500000	3.000000	32.000000	0.500000	141.000000	10.000000	564.000000	1247.000000	2146.500000	12.000000	5.000000	11.000000
75%	1615.250000	2.200000	7.000000	48.000000	0.800000	170.000000	15.000000	947.250000	1633.000000	3064.500000	16.000000	9.000000	16.000000
max	1998.000000	3.000000	19.000000	64.000000	1.000000	200.000000	20.000000	1960.000000	1998.000000	3998.000000	19.000000	18.000000	20.000000

Kolumny dyskretne:

	blue	dual_sim	four_g	n_cores	three_g	touch_screen	wifi	price_range
count	2000.0000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000
mean	0.4950	0.509500	0.521500	4.520500	0.761500	0.503000	0.507000	1.500000
std	0.5001	0.500035	0.499662	2.287837	0.426273	0.500116	0.500076	1.118314
min	0.0000	0.000000	0.000000	1.000000	0.000000	0.000000	0.000000	0.000000
25%	0.0000	0.000000	0.000000	3.000000	1.000000	0.000000	0.000000	0.750000
50%	0.0000	1.000000	1.000000	4.000000	1.000000	1.000000	1.000000	1.500000
75%	1.0000	1.000000	1.000000	7.000000	1.000000	1.000000	1.000000	2.250000
max	1.0000	1.000000	1.000000	8.000000	1.000000	1.000000	1.000000	3.000000

Rysunek 5. Opis kolumn ciągłych i dyskretnych

Wnioski na podstawie uzyskanych informacji:

- Występują minimalne wartości równe 0 dla kolumn ciągłych ‘fc’ i ‘pc’ - te wartości oznaczają, że telefon nie jest wyposażony w przedni lub główny aparat.

- Występują minimalne wartości równe 0 dla kolumny ciągłej ‘px_height’ - kolumna ta wskazuje wysokość rozdzielczości pikseli, więc naszym zdaniem nie powinna wynosić 0, musimy sprawdzić wiersze, które mają taką wartość - zostanie to rozwiązane w kolejnym etapie projektu.

- Występują minimalne wartości równe 0 dla kolumny ciągłej ‘sc_w’ - kolumna ta wskazuje szerokość ekranu w centymetrach, więc naszym zdaniem nie powinna wynosić 0, musimy sprawdzić wiersze, które mają taką wartość - zostanie to rozwiązane w kolejnym etapie projektu.

W analizie danych ważnym celem jest sprawdzanie występowania wartości odstających. Na podstawie informacji z funkcji mean, std, min, max możemy dostrzec dużą różnice między średnią wartością, a maksymalną dla kolumny fc. Z tego powodu podejrzewamy, że w zbiorze mogą znajdować się outliers, więc przeanalizujemy zbiory w celu ich znalezienia z wykorzystaniem z-score.

```
def find_outliers(df):
    outliers_dict = {}
    for col in df.columns:
        z = np.abs(stats.zscore(df[col])) # obliczenie z-score
        outliers = df[z > 3] # sprawdzenie czy dana wartość przekracza z-score 3
        if outliers.shape[0] > 0: # sprawdzenie czy mamy jakiekolwiek wartosci odstajace dla danej kolumny
            outliers_dict[col] = outliers # zapisanie wartosci do slownika
    return outliers_dict
```

Rysunek 6. Kod funkcji znajdującej outliers

```
print("Sprawdzenie outliers dla zbioru treningowego:")
outliers = find_outliers(df_train)

for col, outliers_df in outliers.items():
    print(f"Wartości odstające dla kolumny '{col}':")
    display(outliers_df)
```

Rysunek 7. Kod wyświetlający outliers

```
Sprawdzenie outliers dla zbioru treningowego:
Wartości odstające dla kolumny 'fc':
```

	battery_power	blue	clock_speed	dual_sim	fc	four_g	int_memory	m_dep	mobile_wt	n_cores	...	px_height	px_width	ram	sc_h	sc_w	talk_time	three_g	touch_screen	wifi	price_range
95	1137	1	1.0	0	18	0	7	1.0	196	3	...	942	1179	3616	13	5	12	1	1	1	3
226	1708	1	2.4	1	18	1	49	0.1	109	1	...	233	517	3388	6	4	16	1	1	1	3
305	1348	0	2.0	0	18	0	52	0.3	98	3	...	1869	1942	955	18	11	7	1	1	1	1
1387	1533	1	1.1	1	18	1	17	0.3	160	4	...	1054	1393	2520	8	2	11	1	0	1	2
1406	1731	1	2.3	1	18	0	60	0.5	171	4	...	142	1039	1220	9	3	20	0	1	0	1
1416	1448	0	0.5	1	18	0	2	0.2	100	5	...	846	1144	593	9	4	18	1	1	1	0
1554	1957	0	1.2	1	18	1	36	0.8	151	2	...	1194	1727	1115	16	2	18	1	0	1	1
1693	695	0	0.5	0	18	1	12	0.6	196	2	...	1649	1829	2855	16	13	7	1	1	1	2
1705	1290	1	1.4	1	19	1	35	0.3	110	4	...	405	742	879	16	2	8	1	0	0	0
1880	1720	0	1.6	0	18	1	2	0.8	188	5	...	334	896	2522	10	5	2	1	0	1	2
1882	591	0	2.1	1	18	1	16	0.5	196	7	...	952	1726	704	14	5	4	1	1	1	0
1888	1544	0	2.4	0	18	1	12	0.1	186	7	...	470	844	489	9	4	2	1	0	1	0

Rysunek 8. Wyświetlone wiersze znalezione jako outliers

Funkcja find_outliers przeszukuje cały podany dataframe (wszystkie kolumny) i za pomocą z-score znajduje wartości, które uznajemy za outliers. Arbitralnie ustawionym przez nas z-score powyżej, którego oceniamy wartość jako outlier ustawiliśmy na 3.

Dla naszego zbioru danych funkcja zwróciła 12 outliers, wszystkie z kolumny fc, na którą wcześniej zwróciliśmy uwagę. W kolejnych etapach będziemy musieli rozwiązać problem z występowaniem outliers.

5) Inżynieria cech

Naszym pierwszym zadaniem jest usunięcie wierszy, których wartości px_height i sc_w są równe 0, gdyż uznajemy te wartości za niepoprawne dane. W tym celu sprawdzamy ile jest takich wartości, abyśmy mogli sprawdzić czy zostały one później poprawnie usunięte (sprawdzenie na podstawie ilości wierszy w zbiorze danych).

```
# Sprawdzenie ilości wierszy z sc_w i px_height równymi 0
print("Kolumna sc_w 0:",(df_train['px_height'] == 0).sum())
print("Kolumna px_height 0:",(df_train['sc_w'] == 0).sum())
print("Obie kolumny 0:",((df_train['px_height'] == 0) & (df_train['sc_w'] == 0)).sum())
```

```
Kolumna sc_w 0: 2
Kolumna px_height 0: 180
Obie kolumny 0: 1
```

Rysunek 9. Kod wyświetlający ilość wierszy dla kolumn podejrzanych o błędne dane

W sumie wierszy, których px_height i sc_w są równe 0 jest 181, więc spodziewamy się, że po usunięciu tych wierszy w naszym zbiorze będzie 2000 - 181 = 1819.

```
# Usunięcie wierszy, w których wartość kolumny 'px_height' wynosi 0
df_train = df_train[df_train['px_height'] != 0]

# Wyświetlenie DataFrame po usunięciu wierszy
print(df_train.shape[0])
```

```
1998
```

```
# Usunięcie wierszy, w których wartość kolumny 'sc_w' wynosi 0
df_train = df_train[df_train['sc_w'] != 0]

# Wyświetlenie DataFrame po usunięciu wierszy
print(df_train.shape[0])
```

```
1819
```

Rysunek 10. Kod usuwający błędne dane i wyświetlający ilość wierszy po usunięciu

Po usunięciu podejrzanych danych ilość wierszy zgadza się.

Kolejnym etapem jest usunięcie outliers, ponieważ mogłyby one spowodować gorszą jakość modelu. Poprzednie usunięte wiersze nie pokrywały się z outliers, więc ilość wierszy, którą spodziewamy się po usunięciu to 1819 - 12 = 1807.

```python
def remove_outliers(df, outliers_dict):
    for col, outliers_df in outliers_dict.items():
        indexes_to_drop = outliers_df.index  # pobranie indeksów wierszy do usunięcia
        df = df.drop(indexes_to_drop)  # usunięcie wierszy z DataFrame
    return df
```

```python
df_train = remove_outliers(df_train, outliers)
print(df_train.shape[0])
```

```
1807
```

Rysunek 11. Kod usuwający outliers ze zbioru oraz wyświetlający ilość wierszy po usunięciu

Po usunięciu outliers ilość wierszy w naszym zbiorze zgadza się.

3. Wizualizacja danych wejściowych

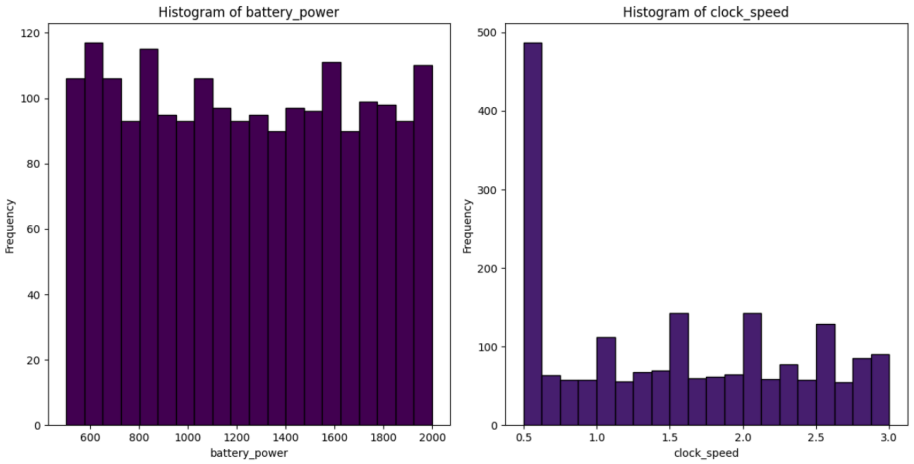
1) Przedstawienie ciągłych danych za pomocą histogramów

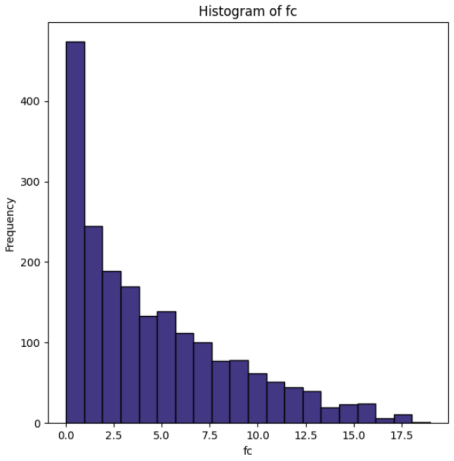
Zastosujemy funkcję plot_histogram, która dla podanego dataframe narysuje histogram dla każdej kolumny, w naszym przypadku przekazuje dataframe, który zawiera tylko kolumny ciągłe.

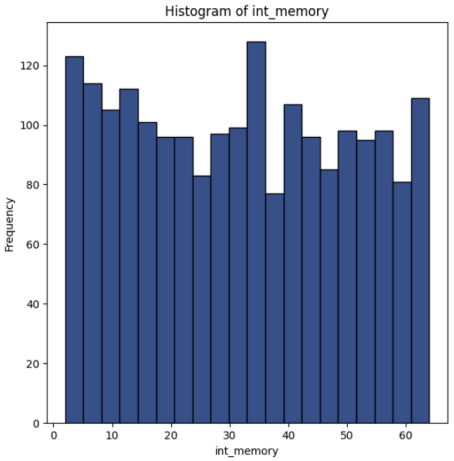
```python
def plot_histograms(df):
    num_cols = len(df.columns)
    num_rows = (num_cols + 1) // 2  # Obliczenie liczby wierszy na podstawie liczby kolumn

    plt.figure(figsize=(12, 6 * num_rows))  # Ustawienie rozmiaru całego wykresu

    # Lista kolorów do użycia w histogramach
    colors = plt.cm.viridis(np.linspace(0, 1, num_cols))

    for i, col in enumerate(df.columns):
        plt.subplot(num_rows, 2, i + 1)  # Ustawienie pozycji histogramu w siatce
        plt.hist(df[col], bins=20, color=colors[i], edgecolor='black')  # Tworzenie histogramu z różnym kolorem
        plt.xlabel(col)  # Dodanie nazwy kolumny jako opisu osi X
        plt.ylabel('Frequency')  # Dodanie opisu osi Y
        plt.title(f'Histogram of {col}')  # Dodanie tytułu wykresu

    plt.tight_layout()  # Zapewnienie odpowiedniego odstępu między subplotami
    plt.show()  # Wyświetlenie wykresu
```

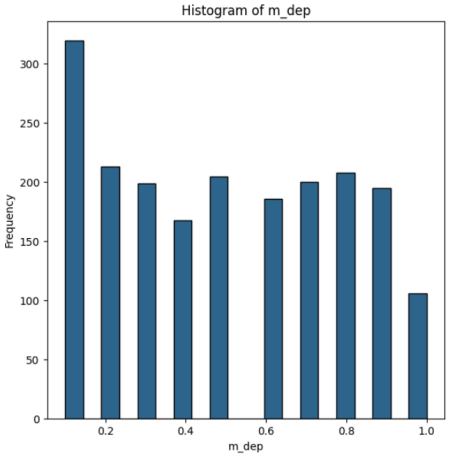
```python
plot_histograms(df_train_c)
```

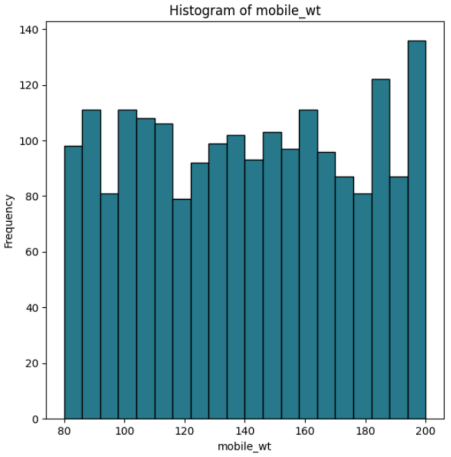
Rysunek 12. Kod funkcji rysujący histogramy dla danych ciągłych

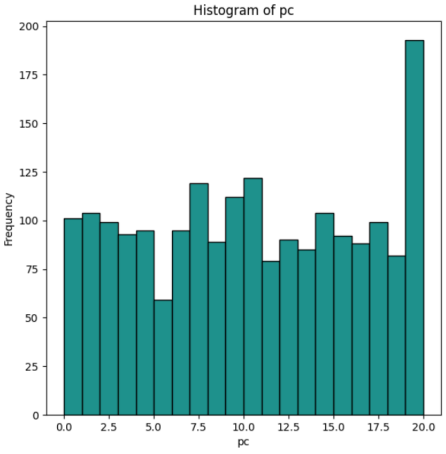


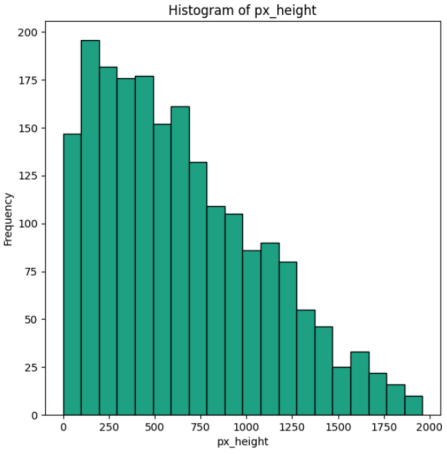

Histogram of fc
Frequency
fc

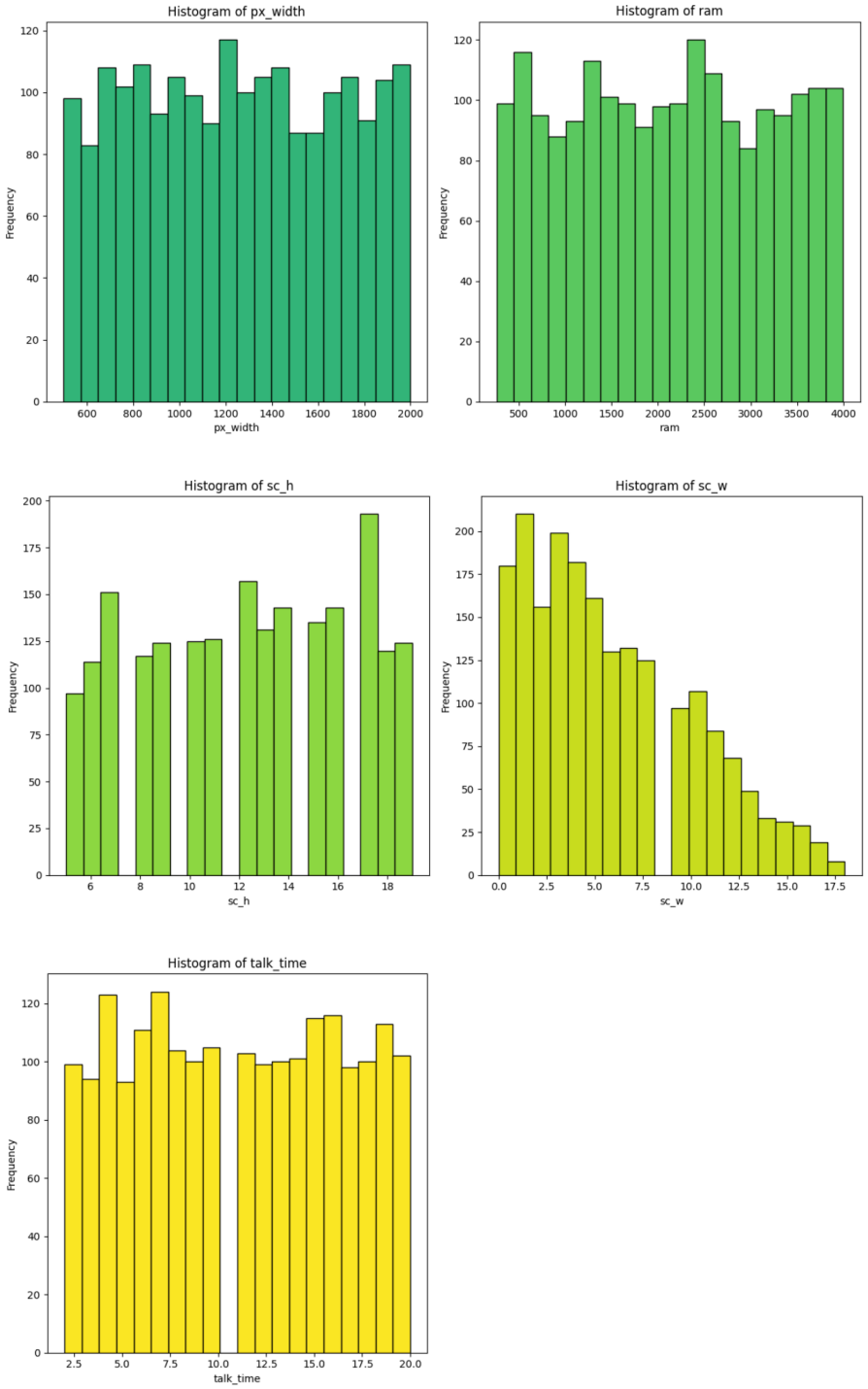

Histogram of int_memory
Frequency
int_memory


Histogram of m_dep
Frequency
m_dep


Histogram of mobile_wt
Frequency
mobile_wt


Histogram of pc
Frequency
pc


Histogram of px_height
Frequency
px_height



Rysunek 13. Histogramy danych ciągłych

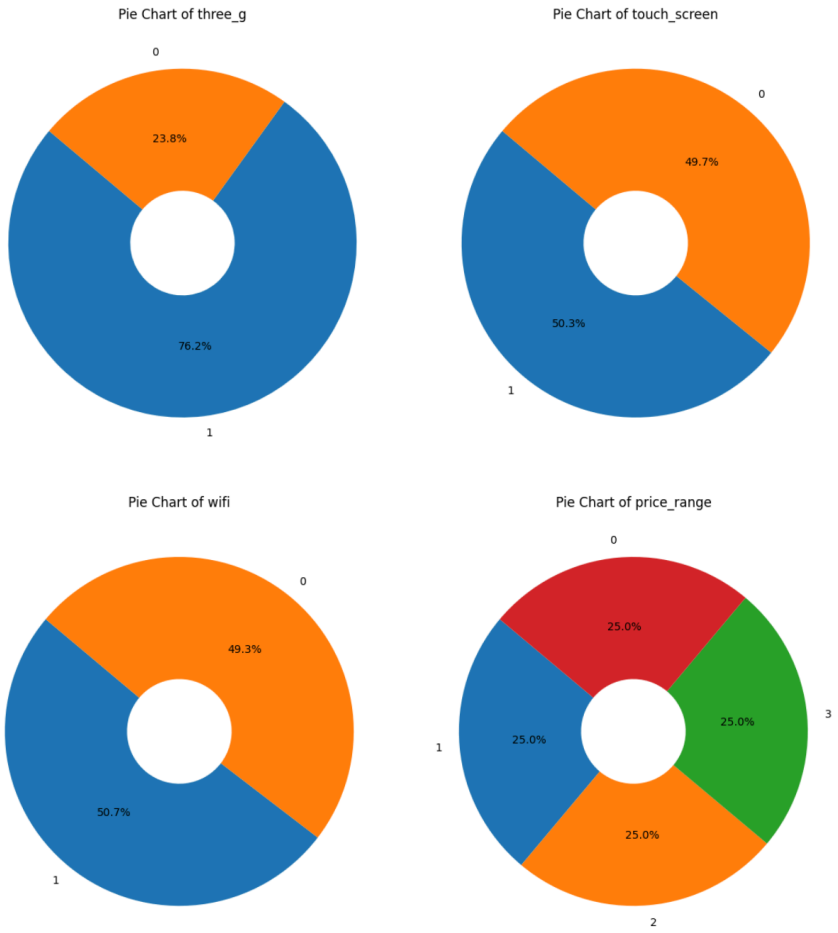
2) Przedstawienie dyskretnych danych za pomocą diagramów kołowych

Zastosujemy funkcję plot_pie_charts, która dla podanego dataframe narysuje diagramy kołowe dla każdej kolumny, w naszym przypadku przekazuje dataframe, który zawiera tylko kolumny dyskretne.

```python
def plot_pie_charts(df):
    num_columns = len(df.columns)
    num_rows = (num_columns + 1) // 2  # Obliczenie liczby wierszy
    fig, axes = plt.subplots(num_rows, 2, figsize=(12, num_rows * 6))  # Ustawienie rozmiaru wykresu

    # Iteracja po wszystkich kolumnach i wierszach
    for i, column_name in enumerate(df.columns):
        row = i // 2
        col = i % 2

        # Wykres kołowy
        value_counts = df[column_name].value_counts()
        ax = axes[row, col]
        ax.pie(value_counts, labels=value_counts.index, autopct='%1.1f%%', startangle=140, wedgeprops=dict(width=0.7))  # Tworzenie wykresu kołowego z pustym środkiem
        ax.set_title(f'Pie Chart of {column_name}')  # Dodanie tytułu wykresu

    # Usunięcie pustych subplotów
    for i in range(num_columns, num_rows * 2):
        fig.delaxes(axes.flatten()[i])

    plt.tight_layout()  # Zapewnienie odpowiedniego odstępu między subplotami
    plt.show()  # Wyświetlenie wykresu
```

```python
# Wywołanie funkcji
plot_pie_charts(df_train_d)
```

Rysunek 14. Kod funkcji rysującej diagramy kołowe dla danych dyskretnych



Rysunek 15. Diagramy kołowe dla danych dyskretnych

Ważnym diagramem jest 'Pie Chart of price_range', gdyż pokazuje on jak dużo procentowo w całym zbiorze mamy etykiet. Gdybyśmy mieli nierówny rozkład musielibyśmy rozwiązać ten problem, ponieważ mogłoby to wpłynąć negatywnie na trenowanie modelu.

Równy rozkład etykiet w połączeniu z histogramami pozwala nam zaobserwować, że pomimo dużej ilości danych o niskiej wartości 'fc' to w zbiorze mamy tyle samo telefonów o wysokiej cenie jak i niskiej. Oznaczać może to, że ta kolumna ma niewielki wpływ na etykietowanie danych. Podobne wnioski możemy wysnuć dla kolumny clock_speed.

4. Budowa modelu

1) Przygotowanie danych

Ze zbioru danych treningowych zapisujemy w osobnym dataframe cechy i w osobnym dataframe targety.

```
# Przygotowanie danych
X = df_train.drop(columns=['price_range'])
y = df_train['price_range']
```

Rysunek 16. Kod dzielący zbiór na X i y

Nasz zbiór danych ma cechy w różnego typu jednostkach z tego powodu normalizujemy dane za pomocą techniki normalizacji z-score.

```
# Normalizacja danych treningowych
X = (X - X.mean()) / X.std()
```

Rysunek 17. Kod normalizujący dane X

Następnie dzielimy cały zbiór danych na część, którą użyjemy do trenowania (80% całości danych) oraz część, którą użyjemy do testowania modelu (20% całości danych).

```
# Podzial danych na zbiór treningowy i testowy
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Rysunek 18. Kod dzielący zbiór na dane treningowe i testowe

Kolejnym etapem jest przekonwertowanie danych do tensorów. Tensor to podstawowa struktura danych w PyTorch, odpowiednik tablic wielowymiarowych w innych bibliotekach. Ta struktura pozwala nam na efektywniejsze wykonywanie operacji.

```
# Konwersja danych do tensorów
X_train_tensor = torch.tensor(X_train.values, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.values, dtype=torch.long)
X_test_tensor = torch.tensor(X_test.values, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test.values, dtype=torch.long)
```

Rysunek 19. Kod konwersji danych na tensory

Dzielimy zbiór testowy na zbiór testowy i walidacyjny, ponieważ na zbiorze testowym będziemy sprawdzać jak dobrze nasz model jest zoptymalizowany do danych testowych, a następnie sprawdzimy jak na danych w pełni nieznanych radzi sobie nasz model.

```
# Przygotowanie danych walidacyjnych z danych testowych
X_test_half1, X_test_half2 = torch.split(X_test_tensor, len(X_test_tensor) // 2)
y_test_half1, y_test_half2 = torch.split(y_test_tensor, len(y_test_tensor) // 2)

# Stworzenie zbiorow danych treningowych i testowych
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_half1, y_test_half1)
val_dataset = TensorDataset(X_test_half2, y_test_half2)
```

Rysunek 20. Kod podziału danych testowych na testowe i walidacyjne

Wykorzystamy również DataLoader, aby pobierać dane w seriach (batch, u nas batch_size ustawiony na 64), dzięki pobieraniu danych w batchach można zoptymalizować zużycie pamięci.

```python
# Wykorzystanie dataLoader do batchowania i shuffle
batch_size = 64
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
val_dataloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
```

Rysunek 21. Kod przekształcenia danych do DataLoader

2) Budowa sieci neuronowej

Stworzymy klasę FeedForwardNN, która będzie definiować model złożony z 3 warstw i funkcji aktywacji Leaky ReLU.

```python
# Tworzenie modelu sieci neuronowej 3 warstwowej, funkcja aktywacji LeakyReLU
class FeedForwardNN(nn.Module):
    def __init__(self, input_size, entry_size, hidden_size, output_size):
        super(FeedForwardNN, self).__init__()
        self.fc1 = nn.Linear(input_size, entry_size)
        self.lrelu1 = nn.LeakyReLU()

        self.fc2 = nn.Linear(entry_size, hidden_size)
        self.lrelu2 = nn.LeakyReLU()

        self.fc3 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        out = self.fc1(x)
        out = self.lrelu1(out)

        out = self.fc2(out)
        out = self.lrelu2(out)

        out = self.fc3(out)
        return out
```

Rysunek 22. Kod definiujący strukturę sieci neuronowej

5. Badanie wpływu liczby neuronów w warstwie ukrytej

1) Opis badania

Przeprowadzimy badanie wpływu liczby neuronów w warstwie ukrytej. Do oceny modelu zastosujemy funkcję accuracy score z biblioteki sklearn na danych testowych. Liczby neuronów, które będą badane to: [1, 2, 4, 8, 20]. Abyśmy lepiej mogli dostrzec efekty różnych ilości neuronów będziemy również testować te badanie dla różnej ilości epok, w ten sposób będziemy mogli dostrzec w jakim tempie model się uczy, liczby epok do testów: [10, 20, 30].

2) Trenowanie modeli

W każdym teście zastosowaliśmy funkcję straty CorssEntropyLoss oraz optymalizator Adama.

```python
results_df = pd.DataFrame(columns=['Hidden Size', 'Epochs', 'Accuracy'])

input_size = X.shape[1]
entry_size = X.shape[1]
num_classes = len(y.unique())
hidden_sizes = [1, 2, 4, 8, 20]
num_epochs_list = [10, 20, 30]

for hidden_size in hidden_sizes:
    for num_epochs in num_epochs_list:
        # Tworzenie modelu
        model = FeedForwardNN(input_size, entry_size, hidden_size, num_classes)

        # Definicja funkcji straty i optymalizatora
        criterion = nn.CrossEntropyLoss()
        optimizer = torch.optim.Adam(model.parameters(), weight_decay=0.001)

        print(f"Testing model with {hidden_size} neurons in hidden layer and {num_epochs} epochs:")

        # Trenowanie modelu
        for epoch in range(num_epochs):
            model.train()
            for inputs, targets in train_dataloader:
                optimizer.zero_grad()
                model_estimation = model(inputs)
                loss = criterion(model_estimation, targets)
                loss.backward()
                optimizer.step()
```

Rysunek 23. Kod trenujący sieć neuronową

3) Ocena modeli

```
# Ocena modelu
true_labels = []
predicted_labels = []

model.eval()

with torch.no_grad():
    for inputs, targets in test_dataloader:
        model_estimation = model(inputs)
        _, predicted = torch.max(model_estimation, 1)
        true_labels.extend(targets.tolist())
        predicted_labels.extend(predicted.tolist())

accuracy = accuracy_score(true_labels, predicted_labels)
```

Rysunek 24. Kod oceny wytrenowanego modelu

4) Analiza wyników

Wyniki zapisaliśmy w dataframe, aby łatwo można było je ocenić.

```
# Dodawanie wyników do DataFrame'u
results_df = pd.concat([results_df, pd.DataFrame({'Hidden Size': [hidden_size], 'Epochs': [num_epochs], 'Accuracy': [accuracy]})], ignore_index=True)

# Wyświetlenie wyników
display(results_df)
```

Rysunek 25. Kod zapisu danych oceny do dataframe

	Hidden Size	Epochs	Accuracy
0	1	10	0.254144
1	1	20	0.480663
2	1	30	0.519337
3	2	10	0.458564
4	2	20	0.657459
5	2	30	0.762431
6	4	10	0.563536
7	4	20	0.795580
8	4	30	0.900552
9	8	10	0.635359
10	8	20	0.906077
11	8	30	0.950276
12	20	10	0.740331
13	20	20	0.911602
14	20	30	0.939227

Rysunek 26. Uzyskane dane w dataframe z dokładności modelu

	Epoki		
Liczba neuronów	10	20	30
1	0.254144	0.480663	0.519337
2	0.458564	0.657459	0.762431
4	0.563536	0.795580	0.900552
8	0.635359	0.906077	0.950276
20	0.740331	0.911602	0.939227

Rysunek 27. Tabela przedstawiająca dane uzyskane w Rysunku 26.

- Drastyczna zmiana wymiarowości nie jest dobrym rozwiązaniem co pokazują wyniki dla 1 i 2 neuronów. Nawet przy większej liczbie epok accuracy modelu nie jest zadowalająca.

- Dla 4 neuronów możemy zauważyć, że przy 30stu epokach dokładność modelu znacząco się poprawiła, dla mniejszej liczby epok 4 neurony nie są wystarczające.

- Wynik dla 30stu epok dla 8 neuronów jest najdokładniejszy, ale trzeba mieć na uwadzę, że to również zależy od losowego wyboru początkowych wag, więc nie powinniśmy brać tego jako wyznacznika. Natomiast możemy zaobserwować, że ze wzrostem ilości neuronów model szybciej się uczy - oznacza to, że musimy wykonać mniejszą liczbę epok, aby uzyskać zadowalającą dokładność.

- Ostatni wynik potwierdza nasz poprzedni wniosek, dlatego dla 20 neuronów widzimy mały wzrost dokładności mimo zwiększenia ilości epok, oznacza to że model został już wytrenowany wcześniej, a dodatkowe epoki nie poprawiają znacząco wyniku - związane jest to z tym, że przez większą ilość neuronów model uczy się szybciej.

Przeprowadziliśmy jeszcze jeden test, w którym wykonaliśmy powyższy kod 10 razy i zapisaliśmy wyniki i obliczyliśmy średnią, aby mieć wyniki mniej uzależnione od losowych wybranych wartości wag.

Average results across all repeats:

	Hidden Size	Epochs	Accuracy
0	1	10	0.340884
1	1	20	0.480110
2	1	30	0.498343
3	2	10	0.437569
4	2	20	0.681768
5	2	30	0.772376
6	4	10	0.554696
7	4	20	0.841989
8	4	30	0.922652
9	8	10	0.643646
10	8	20	0.900000
11	8	30	0.929834
12	20	10	0.797790
13	20	20	0.904972
14	20	30	0.923204

Rysunek 28. Uzyskane dane średniej dokładności w dataframe

Tym razem obliczyliśmy również standardowy rozrzut za pomocą, którego możemy zaobserwować jak nasze poszczególne testy i wyniki uzyskane przez nie są od siebie oddalone.

	Hidden Size	Epochs	Std
0	1	10	0.098833
1	1	20	0.099973
2	1	30	0.068799
3	2	10	0.129774
4	2	20	0.127782
5	2	30	0.082639
6	4	10	0.072561
7	4	20	0.081290
8	4	30	0.017078
9	8	10	0.053264
10	8	20	0.013645
11	8	30	0.009762
12	20	10	0.027587
13	20	20	0.014916
14	20	30	0.019041

Rysunek 29. Uzyskane dane rozrzutu standardowego w dataframe

	Epoki		
Liczba neuronów	10	20	30
1	0.340884	0.480110	0.498343
2	0.437569	0.681768	0.772376
4	0.554696	0.841989	0.922652
8	0.643646	0.900000	0.929834
20	0.797790	0.904972	0.923204

Rysunek 30. Tabela przedstawiająca dane uzyskane w Rysunku 28.

	Epoki		
Liczba neuronów	10	20	30
1	0.098833	0.099973	0.068799
2	0.129774	0.127782	0.082639
4	0.072561	0.081290	0.017078
8	0.053264	0.013645	0.009762
20	0.027587	0.014916	0.019041

Rysunek 31. Tabela przedstawiająca dane uzyskane w Rysunku 29.

Ciekawą obserwacją jest, że 20 neuronów i 8 neuronów przy 20 i 30stu epokach dają bardzo porównywalne wyniki. Można powiedzieć, że większa ilość neuronów jest pomocna kiedy nie chcemy wykonywać dużej ilości epok, ponieważ dla 10ciu epok mimo że dokładność nie jest zadowalająca to dla 20 neuronów jest znacznie lepsza niż dla 8 neuronów.

Możemy zauważyć, że rozrzut standardowy dla małej ilości neuronów jest stosunkowo duży co może sugerować mniejszą stabilność modelu, również interesujące jest, że dla 8 neuronów otrzymujemy mniejszy rozrzut standardowy niż dla 20 neuronów co może sugerować, że ta ilość neuronów jest bardziej stabilna.

6. Badanie wpływu optymalizatorów: SGD, RMSprop, Adam dla różnych wartości parametru: learning_rate

1) Opis optymalizatorów

Stochastic Gradient Descent (SGD) to jeden z najprostszych i najbardziej podstawowych algorytmów optymalizacji gradientowej. Metoda Gradient Descent polega na aktualizacji wag modelu w kierunku przeciwnym do gradientu funkcji straty względem tych wag. Celem jest minimalizacja funkcji straty poprzez schodzenie wzdłuż gradientu w kierunku minimum. W SGD należy dostosować parametr szybkości uczenia się (learning rate), który kontroluje wielkość kroku aktualizacji wag modelu. W SGD, gradient jest obliczany na podstawie losowo wybranej próbki danych treningowych, co przyspiesza proces uczenia i pomaga uniknąć zbytniego dopasowania do konkretnych przykładów.

Root Mean Square Propagation (RMSprop) wykorzystuje adaptacyjny współczynnik skalowania gradientu dla każdego parametru. Nie zawiera składowej momentum, która miałaby zapamiętywać historyczne gradienty. Oznacza to, że aktualizacje wag modelu są oparte tylko na bieżących gradientach. RMSprop pomaga w stabilizacji procesu uczenia poprzez skalowanie gradientów, co jest szczególnie przydatne w problemach, gdzie skale gradientów różnią się znacząco.

Adaptive Moment Estimation (Adam) to metoda optymalizacji gradientowej, która łączy w sobie zalety kilku innych popularnych optymalizatorów. Moment pierwszego rzędu (średnia ruchoma gradientu) jak i moment drugiego rzędu (średnia ruchoma kwadratu gradientu) są adaptacyjnie obliczane dla każdego parametru modelu. Adam stosuje bias-korektę dla estymat momentów pierwszego i drugiego rzędu, co pomaga w poprawie ich wartości początkowych i stabilizuje proces uczenia.

2) Opis badania

Przeprowadzimy badanie 3 optymalizatorów dla sieci neuronowej feed forward zbudowanej z 3 warstw, warstwa wejściowa 20 neuronów, ukryta 8 neuronów, wyjściowa 4 neurony. Do oceny modelu zastosujemy funkcję accuracy score z biblioteki sklearn na danych testowych. Optymalizatory biorące udział w badaniu to wspomniane już wyżej: SGD, RMSprop, Adam. Zastosujemy 3 różne learning_rate dla każdego optymalizatora.

2) Trenowanie modeli

```
hidden_size = 8
num_epochs = 30

learning_rates = [0.1, 0.01, 0.001]

# Lista optymalizatorów do porównania
optimizers = ['SGD', 'RMSprop', 'Adam']
for lr in learning_rates:
    for optimizer_name in optimizers:
        # Tworzenie modelu
        model = FeedForwardNN(input_size, entry_size, hidden_size, num_classes)

        # Definicja funkcji straty
        criterion = nn.CrossEntropyLoss()

        # Wybór optymalizatora
        if optimizer_name == 'SGD':
            optimizer = torch.optim.SGD(model.parameters(), lr=lr)
        elif optimizer_name == 'RMSprop':
            optimizer = torch.optim.RMSprop(model.parameters(), lr=lr)
        elif optimizer_name == 'Adam':
            optimizer = torch.optim.Adam(model.parameters(), lr=lr, weight_decay=0.001)

        print(f"Testing model with {optimizer_name} optimizer and learning rate {lr}:")

        # Trenowanie modelu
        for epoch in range(num_epochs):
            model.train()
            for inputs, targets in train_dataloader:
                optimizer.zero_grad()
                model_estimation = model(inputs)
                loss = criterion(model_estimation, targets)
                loss.backward()
                optimizer.step()
```

Rysunek 32. Kod trenowania 3 optymalizatorów dla 3 learning rate

3) Ocena modeli

```
        # Trenowanie modelu
        for epoch in range(num_epochs):
            model.train()
            for inputs, targets in train_dataloader:
                optimizer.zero_grad()
                model_estimation = model(inputs)
                loss = criterion(model_estimation, targets)
                loss.backward()
                optimizer.step()

        # Ocena modelu
        true_labels = []
        predicted_labels = []

        model.eval()

        with torch.no_grad():
            for inputs, targets in test_dataloader:
                model_estimation = model(inputs)
                _, predicted = torch.max(model_estimation, 1)
                true_labels.extend(targets.tolist())
                predicted_labels.extend(predicted.tolist())

        accuracy = accuracy_score(true_labels, predicted_labels)

        # Dodawanie wyników do DataFrame'u
        results_o_df = pd.concat([results_o_df, pd.DataFrame({'Optimizer': [optimizer_name], 'Learning Rate': [lr], 'Accuracy': [accuracy]})], ignore_index=Tru

# Wyświetlenie wyników
display(results_o_df)
```

Rysunek 33. Kod oceny 3 optymalizatorów dla 3 learning rate

4) Analiza wyników

Wyniki zapisaliśmy w dataframe, aby łatwo można było je ocenić.

	Optimizer	Learning Rate	Accuracy
0	SGD	0.100	0.878453
1	RMSprop	0.100	0.883978
2	Adam	0.100	0.917127
3	SGD	0.010	0.497238
4	RMSprop	0.010	0.906077
5	Adam	0.010	0.922652
6	SGD	0.001	0.243094
7	RMSprop	0.001	0.911602
8	Adam	0.001	0.928177

Rysunek 34. Wyniki badania 3 optymalizatorów dla 3 learning rate

- Optymalizator Adam osiągnął najwyższą dokładność dla wszystkich badanych wartości szybkości uczenia się, co sugeruje jego skuteczność w procesie uczenia się.

- RMSprop również osiągnął wysoką dokładność, szczególnie dla mniejszych wartości szybkości uczenia się.

- Optymalizator SGD miał trudności w osiągnięciu wysokiej dokładności, szczególnie dla mniejszych wartości szybkości uczenia się, co sugeruje jego ograniczoną skuteczność w porównaniu do RMSprop i Adam dla tego konkretnego zadania i zbioru danych.

7. Badanie wpływu parametrów dla różnych optymalizatorów

1) Opis badania

Przeprowadzimy badanie, w którym chcemy sprawdzić wpływ dostosowania opcjonalnych parametrów dla poszczególnych optymalizatorów: SGD, RMSprop, Adam. W tym celu zastosujemy iloczyn kartezjański dla wybranych przez nas parametrów do różnych optymalizatorów. W celu uzyskania wyników, które są wiarygodne dla każdego połączenia wykonaliśmy 10 badań, z których następnie wyciągnęliśmy średnią.

2) Optymalizator SGD

Testowane parametry dla SGD:
weight_decays = [0.0, 0.01, 0.001]
momentum_values = [0.0, 0.5, 0.9]
dampening_values = [0.0, 0.1, 0.2]

Na tych parametrach wykonamy iloczyn kartezjański i obliczymy dla każdego połączenia dokładność modelu. Pierwsze parametry zostały tak dobrane, aby były one domyślnymi wartościami funkcji optymalizatora

a) Kod programu

Wykorzystujemy bibliotekę itertools, aby otrzymać iloczyn kartezjański parametrów.

```
# Parametry do testowania
weight_decays = [0.0, 0.01, 0.001]
momentum_values = [0.0, 0.5, 0.9]
dampening_values = [0.0, 0.1, 0.2]

# Tworzenie wszystkich możliwych kombinacji parametrów
param_combinations = list(itertools.product(weight_decays, momentum_values, dampening_values))

# Pętla po kombinacjach parametrów
for weight_decay, momentum, dampening in param_combinations:
    accuracies = []  # Lista do przechowywania dokładności dla danej kombinacji parametrów
    # Powtarzanie treningu 10 razy
    for _ in range(10):
        # Tworzenie modelu
        model = FeedForwardNN(input_size, entry_size, hidden_size, num_classes)

        # Definicja funkcji straty
        criterion = nn.CrossEntropyLoss()

        # Definicja optymalizatora SGD z odpowiednimi parametrami
        optimizer = torch.optim.SGD(model.parameters(), momentum=momentum, dampening=dampening, weight_decay=weight_decay)

        # Trenowanie modelu
        for epoch in range(num_epochs):
            model.train()
            for inputs, targets in train_dataloader:
                optimizer.zero_grad()
                model_estimation = model(inputs)
                loss = criterion(model_estimation, targets)
                loss.backward()
                optimizer.step()
```

Rysunek 35. Kod trenowania optymalizatora SGD dla różnych parametrów

```
        # Ocena modelu
        true_labels = []
        predicted_labels = []

        model.eval()

        with torch.no_grad():
            for inputs, targets in test_dataloader:
                model_estimation = model(inputs)
                _, predicted = torch.max(model_estimation, 1)
                true_labels.extend(targets.tolist())
                predicted_labels.extend(predicted.tolist())

        accuracy = accuracy_score(true_labels, predicted_labels)
        accuracies.append(accuracy)  # Dodanie dokładności dla danego treningu

    # Obliczenie średniej dokładności dla danej kombinacji parametrów
    avg_accuracy = sum(accuracies) / len(accuracies)

    # Dodawanie wyników do DataFrame'u
    results_df = pd.concat([results_df, pd.DataFrame({
        'Weight Decay': [weight_decay],
        'Momentum': [momentum],
        'Dampening': [dampening],
        'Accuracy': [avg_accuracy]
    })], ignore_index=True)

# Wyświetlenie wyników
display(results_df)
```

Rysunek 36. Kod oceny optymalizatora SGD dla różnych parametrów

b) Wyniki

	Weight Decay	Momentum	Dampening	Accuracy
0	0.000	0.0	0.0	0.265746
1	0.000	0.0	0.1	0.256354
2	0.000	0.0	0.2	0.246961
3	0.000	0.5	0.0	0.266851
4	0.000	0.5	0.1	0.249171
5	0.000	0.5	0.2	0.259669
6	0.000	0.9	0.0	0.427072

Rysunek 37. Wyniki badania SGD dla różnych parametrów (skrócone)

c) Analiza wyników

Przekształciliśmy wyniki w formie poniższej tabeli i posortowaliśmy ją po Accuracy.

Weight Decay	Momentum	Dampening	Accuracy
0.001	0.9	0.1	0.452486
0.000	0.9	0.0	0.427072
0.001	0.9	0.0	0.423204
0.001	0.9	0.2	0.420994
0.010	0.9	0.0	0.411050
0.000	0.9	0.2	0.409945
0.000	0.9	0.1	0.400000
0.010	0.9	0.1	0.391160
0.010	0.9	0.2	0.372376
0.010	0.0	0.0	0.281215
0.010	0.5	0.1	0.275138
0.001	0.5	0.1	0.271823
0.010	0.5	0.2	0.269613
0.000	0.5	0.0	0.266851
0.001	0.0	0.0	0.266298
0.000	0.0	0.0	0.265746
0.001	0.5	0.0	0.265193
0.000	0.5	0.2	0.259669
0.001	0.0	0.2	0.259116

0.000	0.0	0.1	0.256354
0.001	0.5	0.2	0.255801
0.000	0.5	0.1	0.249171
0.010	0.5	0.0	0.249171
0.010	0.0	0.2	0.248619
0.000	0.0	0.2	0.246961
0.001	0.0	0.1	0.242541
0.010	0.0	0.1	0.240884

Rysunek 37. Tabela przedstawiająca wyniki badania SGD dla różnych parametrów

d) Wnioski

Warto zauważyć, że zaznaczony na zielono wynik jest uzyskany przez domyślne wartości parametrów. Możemy zobaczyć dużą różnice między tym wynikiem ok. 27%, a najlepszym wynikiem dającym ok. 45%. Na podstawie tego wysuwa się nam wniosek, że poprawne ustawienie parametrów wpływa znacząco na wynik. Interesujące jest również, że model z domyślnymi parametrami nie jest najgorszym na liście, co pokazuje również, że źle dobrane wartości dla parametrów mogą pogorszyć wynik.

Dla naszego problemu i modelu największy wpływ na dokładność wykazuje parametr momentum, ponieważ dla jego wartości 0.9 otrzymujemy najlepsze wyniki.

3) Optymalizator RMSprop

Testowane parametry dla RMSprop:
weight_decays = [0.0, 0.01, 0.001]
alphas = [0.99, 0.5, 0.1]
momentum_values = [0.0, 0.5, 0.9]

Badanie zostanie przeprowadzone analogicznie do badania optymalizatora sgd, to znaczy że mamy domyślne parametry oraz wykonamy iloczyn kartezjański.

a) Kod programu

```
# Parametry do testowania
weight_decays = [0.0, 0.01, 0.001]
alphas = [0.99, 0.5, 0.1]
momentum_values = [0.0, 0.5, 0.9]

# Tworzenie wszystkich możliwych kombinacji parametrów
param_combinations = list(itertools.product(weight_decays, alphas, momentum_values))

# Pętla po kombinacjach parametrów
for weight_decay, alpha, momentum in param_combinations:
    accuracies = []  # Lista do przechowywania dokładności dla danej kombinacji parametrów
    # Powtarzanie treningu 10 razy
    for _ in range(10):
        # Tworzenie modelu
        model = FeedForwardNN(input_size, entry_size, hidden_size, num_classes)

        # Definicja funkcji straty
        criterion = nn.CrossEntropyLoss()

        # Definicja optymalizatora RMSprop z odpowiednimi parametrami
        optimizer = torch.optim.RMSprop(model.parameters(), alpha=alpha, momentum=momentum, weight_decay=weight_decay)

        # Trenowanie modelu
        for epoch in range(num_epochs):
            model.train()
            for inputs, targets in train_dataloader:
                optimizer.zero_grad()
                model_estimation = model(inputs)
                loss = criterion(model_estimation, targets)
                loss.backward()
                optimizer.step()
```

Rysunek 38. Kod trenowania optymalizatora RMSprop dla różnych parametrów

```
        # Ocena modelu
        true_labels = []
        predicted_labels = []

        model.eval()

        with torch.no_grad():
            for inputs, targets in test_dataloader:
                model_estimation = model(inputs)
                _, predicted = torch.max(model_estimation, 1)
                true_labels.extend(targets.tolist())
                predicted_labels.extend(predicted.tolist())

        accuracy = accuracy_score(true_labels, predicted_labels)
        accuracies.append(accuracy)  # Dodanie dokładności dla danego treningu

    # Obliczenie średniej dokładności dla danej kombinacji parametrów
    avg_accuracy = sum(accuracies) / len(accuracies)

    # Dodawanie wyników do DataFrame'u
    results_df = pd.concat([results_df, pd.DataFrame({
        'Weight Decay': [weight_decay],
        'Alpha': [alpha],
        'Momentum': [momentum],
        'Accuracy': [avg_accuracy]
    })], ignore_index=True)

# Wyświetlenie wyników
display(results_df)
```

Rysunek 39. Kod oceny optymalizatora RMSprop dla różnych parametrów

b) Wyniki

	Weight Decay	Alpha	Momentum	Accuracy
0	0.000	0.99	0.0	0.901105
1	0.000	0.99	0.5	0.883978
2	0.000	0.99	0.9	0.906077
3	0.000	0.50	0.0	0.911602
4	0.000	0.50	0.5	0.895028
5	0.000	0.50	0.9	0.867403
6	0.000	0.10	0.0	0.907735

Rysunek 40. Wyniki badania optymalizatora RMSprop dla różnych parametrów (skrócone)

c) Analiza wyników

Przekształciliśmy wyniki w formie poniższej tabeli i posortowaliśmy ją po Accuracy.

Weight Decay	Alpha	Momentum	Accuracy
0.010	0.99	0.5	0.922099
0.010	0.50	0.0	0.919890
0.010	0.10	0.0	0.918232
0.010	0.99	0.0	0.915470
0.010	0.50	0.5	0.914365
0.001	0.10	0.0	0.914365
0.000	0.50	0.0	0.911602
0.010	0.10	0.5	0.911602
0.000	0.10	0.0	0.907735
0.000	0.99	0.9	0.906077
0.010	0.99	0.9	0.905525
0.000	0.10	0.5	0.904972
0.001	0.50	0.0	0.903867
0.001	0.99	0.0	0.902762
0.000	0.99	0.0	0.901105
0.001	0.50	0.5	0.900000
0.001	0.10	0.5	0.898343
0.000	0.50	0.5	0.895028

0.010	0.50	0.9	0.886188
0.001	0.99	0.9	0.885635
0.001	0.99	0.5	0.885083
0.000	0.99	0.5	0.883978
0.001	0.50	0.9	0.876796
0.001	0.10	0.9	0.869613
0.000	0.50	0.9	0.867403
0.010	0.10	0.9	0.867403
0.000	0.10	0.9	0.864641

Rysunek 41. Tabela przedstawiająca wyniki badania RMSprop dla różnych parametrów

d) Wnioski

Podobnie jak w poprzednim badaniu wynik zaznaczony na zielono jest otrzymany przez domyślne wartości parametrów. Tak samo jak w poprzednim badaniu możemy zauważyć duży wpływ na dokładność modelu wartości parametrów. Czynnikiem, który ma największy wpływ na dokładność jest momentum, niskie wartości tego czynnika wydają się dawać lepsze rezultaty, a gorszych wyników posiada wysoki współczynnik momentum.

Optymalizator SGD w porównaniu do RMSprop jest znacznie łatwiejszym algorytmem, dlatego tam łatwiej jest dobrać odpowiednie parametry. W RMSprop znacznie trudniej jest dobrać dobre wartości parametrów, co widać w wynikach, że różne połączenia wartości dają różne wyniki - ciężko wskazać schemat w ustawianiu wartości.

4) Optymalizator Adam

Testowane parametry dla Adam:
weight_decays = [0.0, 0.01, 0.001]
betas_options = [(0.9, 0.999), (0.5, 0.999), (0.5, 0.9)]
amsgrad_options = [False, True]

Badanie zostanie przeprowadzone analogicznie do badania optymalizatora sgd i rmsprop, to znaczy że mamy domyślne parametry oraz wykonamy iloczyn kartezjański.

a) Kod programu

```
# Parametry do testowania
weight_decays = [0.0, 0.01, 0.001]
betas_options = [(0.9, 0.999), (0.5, 0.999), (0.5, 0.9)]
amsgrad_options = [False, True]

# Tworzenie wszystkich możliwych kombinacji parametrów
param_combinations = list(itertools.product(weight_decays, betas_options, amsgrad_options))

# Pętla po kombinacjach parametrów
for weight_decay, betas, amsgrad in param_combinations:
    accuracies = []  # Lista do przechowywania dokładności dla danej kombinacji parametrów
    # Powtarzanie treningu 10 razy
    for _ in range(10):
        # Tworzenie modelu
        model = FeedForwardNN(input_size, entry_size, hidden_size, num_classes)

        # Definicja funkcji straty
        criterion = nn.CrossEntropyLoss()

        # Definicja optymalizatora Adam z odpowiednimi parametrami
        optimizer = torch.optim.Adam(model.parameters(), weight_decay=weight_decay, betas=betas, amsgrad=amsgrad)

        # Trenowanie modelu
        for epoch in range(num_epochs):
            model.train()
            for inputs, targets in train_dataloader:
                optimizer.zero_grad()
                model_estimation = model(inputs)
                loss = criterion(model_estimation, targets)
                loss.backward()
                optimizer.step()
```

Rysunek 42. Kod trenowania optymalizatora Adam dla różnych parametrów

```
        # Ocena modelu
        true_labels = []
        predicted_labels = []

        model.eval()

        with torch.no_grad():
            for inputs, targets in test_dataloader:
                model_estimation = model(inputs)
                _, predicted = torch.max(model_estimation, 1)
                true_labels.extend(targets.tolist())
                predicted_labels.extend(predicted.tolist())

        accuracy = accuracy_score(true_labels, predicted_labels)
        accuracies.append(accuracy)  # Dodanie dokładności dla danego treningu

    # Obliczenie średniej dokładności dla danej kombinacji parametrów
    avg_accuracy = sum(accuracies) / len(accuracies)

    # Dodawanie wyników do DataFrame'u
    results_df = pd.concat([results_df, pd.DataFrame({
        'Weight Decay': [weight_decay],
        'Betas': [betas],
        'Amsgrad': [amsgrad],
        'Accuracy': [avg_accuracy]
    })], ignore_index=True)

# Wyświetlenie wyników
display(results_df)
```

Rysunek 43. Kod oceny optymalizatora Adam dla różnych parametrów

b) Wyniki

	Weight Decay	Betas	Amsgrad	Accuracy
0	0.000	(0.9, 0.999)	False	0.923757
1	0.000	(0.9, 0.999)	True	0.917680
2	0.000	(0.5, 0.999)	False	0.921547
3	0.000	(0.5, 0.999)	True	0.927624
4	0.000	(0.5, 0.9)	False	0.911050
5	0.000	(0.5, 0.9)	True	0.888950
6	0.010	(0.9, 0.999)	False	0.947514

Rysunek 44. Wyniki badania optymalizatora Adam dla różnych parametrów (skrócone)

c) Analiza wyników

Przekształciliśmy wyniki w formie poniższej tabeli i posortowaliśmy ją po Accuracy.

Weight Decay	Betas	Amsgrad	Accuracy
0.010	(0.5, 0.999)	TRUE	0.954144
0.010	(0.5, 0.999)	FALSE	0.949724
0.010	(0.9, 0.999)	FALSE	0.947514
0.010	(0.9, 0.999)	TRUE	0.947514
0.001	(0.5, 0.999)	TRUE	0.930387
0.001	(0.5, 0.999)	FALSE	0.929282
0.010	(0.5, 0.9)	FALSE	0.928729
0.001	(0.9, 0.999)	TRUE	0.928729
0.000	(0.5, 0.999)	TRUE	0.927624
0.001	(0.9, 0.999)	FALSE	0.925414
0.000	(0.9, 0.999)	FALSE	0.923757
0.000	(0.5, 0.999)	FALSE	0.921547
0.000	(0.9, 0.999)	TRUE	0.917680
0.001	(0.5, 0.9)	FALSE	0.912155
0.000	(0.5, 0.9)	FALSE	0.911050
0.010	(0.5, 0.9)	TRUE	0.902762
0.001	(0.5, 0.9)	TRUE	0.893923
0.000	(0.5, 0.9)	TRUE	0.888950

Rysunek 45. Tabela przedstawiająca wyniki badania Adam dla różnych parametrów

d) Wnioski

Podobnie jak w poprzednim badaniu wynik zaznaczony na zielono jest otrzymany przez domyślne wartości parametrów. Tak samo jak w poprzednim badaniu możemy zauważyć duży wpływ na dokładność modelu wartości parametrów.

W naszym badaniu dla naszego modelu czynnikiem mającym największy wpływ wydaje się być weight decay oraz betas. Weight decay dla wartości 0.01 dawał najlepsze wyniki z parametrami betas (0.5, 0.999).

Niski wynik dla parametrów betas (0.5, 0.9) może sugerować, że nawet mała zmiana drugiego czynnika betas może znacząco pogorszyć jakość modelu.

8. Test najlepszego modelu na danych walidacyjnych

1) Najlepszy model

Na podstawie wyników z poprzednich badań najlepszym testowanym modelem wybieramy model z 8 neuronami w warstwie ukrytej oraz z optymalizatorem Adam'a, który podczas badań okazał się najbardziej stabilny - wybieramy model z domyślnymi pozostałymi parametrami.

2) Algorytm walidacyjny

Sprawdzamy czy nasz model jest faktycznie dobry, czy został zoptymalizowany pod dane testowe, w tym celu użyjemy wcześniej zdefiniowanego zbioru walidacyjnego, którego wcześniej nie używaliśmy podczas ocen.

```
model.eval()

with torch.no_grad():
    for inputs, targets in test_dataloader:
        model_estimation = model(inputs)
        _, predicted = torch.max(model_estimation, 1)
        true_labels.extend(targets.tolist())
        predicted_labels.extend(predicted.tolist())

accuracy_test = accuracy_score(true_labels, predicted_labels)

# Ocena modelu
true_labels = []
predicted_labels = []

model.eval()

with torch.no_grad():
    for inputs, targets in val_dataloader:
        model_estimation = model(inputs)
        _, predicted = torch.max(model_estimation, 1)
        true_labels.extend(targets.tolist())
        predicted_labels.extend(predicted.tolist())

accuracy_val = accuracy_score(true_labels, predicted_labels)

# Dodawanie wyników do DataFrame'u
results_final_df = pd.concat([results_final_df, pd.DataFrame({'Test Accuracy': [accuracy_test], 'Val Accuracy': [accuracy_val]})], ignore_index=True)

# Wyświetlenie wyników
display(results_final_df)
```

Rysunek 46. Kod oceny modelu za pomocą danych walidacyjnych

	Test Accuracy	Val Accuracy
0	0.917127	0.944751

Rysunek 47. Wynik oceny modelu za pomocą danych walidacyjnych

Model na danych wcześniej nie widzianych również radzi sobie bardzo dobrze, więc możemy stwierdzić, że jest dobrze wytrenowany.

9. Wnioski

Inżynieria cech to: analiza danych, oczyszczenie zbioru z wartości odstających i niepoprawnych danych oraz odpowiednia normalizacja są niezbędne do uzyskania dobrych wyników modelu. Dokładność i rzetelność danych mają istotny wpływ na skuteczność uczenia maszynowego.

Architektura sieci: badanie wpływu liczby neuronów w warstwie ukrytej oraz innych parametrów architektury sieci pozwala na zrozumienie, jak zbudować model optymalnie dopasowany do konkretnego zadania. Złożoność sieci i liczba neuronów mają bezpośredni wpływ na zdolność modelu do generalizacji i dokładności predykcji.

Wybór optymalizatora: porównanie różnych optymalizatorów (takich jak SGD, RMSprop, Adam) pozwala na wybór najlepiej dopasowanego do problemu. Skuteczność optymalizacji ma istotny wpływ na tempo uczenia się modelu i jego zdolność do osiągnięcia wysokiej dokładności.

Wpływ parametrów: badane optymalizatory mają różne parametry, które mają duży wpływ na jakość trenowanego modelu. Dobranie odpowiednich parametrów jest złożonym zadaniem, które wymaga bardzo dobrego zrozumienia i doświadczenia w budowie modeli neuronowych.