Maxym Besarab 164042

Lizaveta Valius 183052

Karyna Lukashova 187714

Chatbot asystent bankowy

Wprowadzenie

Projekt chatbota AI został opracowany przy użyciu biblioteki nltk do przetwarzania tekstu oraz biblioteki keras do tworzenia i trenowania modelu sieci neuronowej. Chatbot ma na celu odpowiadać na pytania użytkowników w czasie rzeczywistym, bazując na zestawie intencji i ich wzorcach.

Instalacja zależności

Aby uruchomić projekt, należy zainstalować następujące zależności:

- nltk: Biblioteka do przetwarzania tekstu w języku naturalnym.
- keras: Biblioteka do tworzenia i trenowania modeli sieci neuronowych.
- numpy: Biblioteka do obliczeń naukowych w języku Python.
- Tkinter: Biblioteka do tworzenia interfejsów graficznych w języku Python.

Projekt składa się z trzech głównych części:

1. Przygotowanie danych i trenowanie modelu: “train_chatbot.py” i „train_chatbot_ADAM” są odpowiedzialne za wczytywanie i przetwarzanie danych, a następnie trenowanie modelu. Dane są oczyszczane, lematyzowane i tokenizowane, a następnie przekształcane w worki słów (bag of words) dla treningu.
2. Definicja Modelu: “Model.py” jest definicją klasy Model, która jest klasą bazową dla wszystkich modeli w projekcie. Model ten jest konstruowany, kompilowany i trenowany za pomocą danych, które zostały przygotowane w pierwszej części kodu.
3. Implementacja Chat Bot: “chatgui.py” jest implementacją interfejsu użytkownika dla chatbota za pomocą biblioteki Tkinter. Chatbot interpretuje pytania użytkownika, przetwarza je i generuje odpowiedzi na podstawie modelu, który został wytrenowany.

Opis pliku „train_chatbot.py”:

Kod przedstawiony poniżej odpowiada za trening Chatbota, czyli proces uczenia modelu na podstawie dostarczonych danych treningowych. Chatbot będzie w stanie udzielać odpowiedzi na pytania użytkowników w zależności od nauczonej wiedzy i intencji.

- Importowanie zależności

Kod rozpoczyna się od importowania niezbędnych bibliotek:

```
import nltk
from nltk.stem import WordNetLemmatizer
import json
import pickle
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import SGD
import random
```

- Przygotowanie danych treningowych

Kod wczytuje dane treningowe z pliku JSON, który zawiera intencje (kategorie) i odpowiadające im wzorce pytań. Następnie tokenizuje wzorce na pojedyncze słowa, lematyzuje je (reprezentuje w formie podstawowej) i tworzy listy unikalnych słów i intencji.

```
lemmatizer = WordNetLemmatizer()
words=[]
classes = []
documents = []
ignore_words = ['?', '!']
data_file = open('intents.json').read()
intents = json.loads(data_file)


for intent in intents['intents']:
    for pattern in intent['patterns']:

        w = nltk.word_tokenize(pattern)
        words.extend(w)
        documents.append((w, intent['tag']))

        if intent['tag'] not in classes:
            classes.append(intent['tag'])

words = [lemmatizer.lemmatize(w.lower()) for w in words if w not in ignore_words]
words = sorted(list(set(words)))
classes = sorted(list(set(classes)))
```

- Przygotowanie danych treningowych w formacie bag-of-words

Kod przetwarza wzorce pytań na wektory "worków słów" (bag-of-words). Dla każdego wzorca tworzony jest wektor, który zawiera wartość 1, jeśli dane słowo występuje w wzorcu, a w przeciwnym przypadku wartość 0. Dodatkowo, tworzone są odpowiednie etykiety dla danych treningowych.

```
training = []
output_empty = [0] * len(classes)
for doc in documents:
    bag = []
    pattern_words = doc[0]
    pattern_words = [lemmatizer.lemmatize(word.lower()) for word in pattern_words]
    for w in words:
        bag.append(1) if w in pattern_words else bag.append(0)

    output_row = list(output_empty)
    output_row[classes.index(doc[1])] = 1

    training.append([bag, output_row])
```

- Trenowanie modelu

Kod przetasowuje dane treningowe i przekształca je na format zrozumiały dla modelu. Następnie tworzony jest model sieci neuronowej. Składa się on z trzech warstw ukrytych: pierwsza warstwa z 128 neuronami, druga warstwa z 64 neuronami, a trzecia warstwa wyjściowa zawiera liczbę neuronów odpowiadającą liczbie intencji. Model jest kompilowany z wykorzystaniem optymalizatora SGD i funkcji straty categorical_crossentropy. Następnie jest trenowany na danych treningowych przez określoną liczbę epok.

```
random.shuffle(training)
training = np.array(training)
train_x = list(training[:,0])
train_y = list(training[:,1])
print("Training data created")

model = Sequential()
model.add(Dense(128, input_shape=(len(train_x[0]),), activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(len(train_y[0]), activation='softmax'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])

hist = model.fit(np.array(train_x), np.array(train_y), epochs=200, batch_size=5, verbose=1)
```

- Zapisanie nauczonego modelu

Po zakończeniu trenowania modelu, kod zapisuje go do pliku "chatbot_model.h5" w celu późniejszego wykorzystania.

```
model.save('chatbot_model.h5', hist)
```

To samo mamy z kodem „train_chatbot_ADAM.py”

Opis pliku „intents.json”:
Plik "intents.json" zawiera definicje intencji, wzorców i odpowiedzi używanych przez asystenta do rozpoznawania i reagowania na użytkownika w kontekście bankowości. Każda intencja reprezentuje określone zapytanie lub komunikat użytkownika, a skojarzone z nimi wzorce i odpowiedzi służą do nauki asystenta w celu udzielenia właściwej odpowiedzi na dane pytanie lub komunikat.

Struktura pliku:

- "intents" (tablica) - zawiera listę wszystkich intencji, wzorców i odpowiedzi.

Dla każdej intencji:

- "tag" (łańcuch znaków) - unikalny identyfikator intencji”
- "patterns" (tablica) - zawiera wzorce, które mogą odpowiadać zapytaniom lub komunikatom użytkownika związanych z daną intencją.
- "responses" (tablica) - zawiera odpowiedzi, które asystent może udzielić na zapytania lub komunikaty użytkownika związane z daną intencją.

```
{
    "intents": [
        {
            "tag": "greeting",
            "patterns": [
                "Hello",
                "Hi",
                "Hey",
                "Good morning"
            ],
            "responses": [
                "Hello! How may I assist you?",
                "Hi there! What can I do for you?",
                "Good morning! How can I help you?"
            ]
        },
```

Opis pliku „CustomSGD.py”:
Ten kod implementuje algorytm SGD-NAG (Stochastic Gradient Descent with Nesterov Accelerated Gradient) w celu treningu modelu. Algorytm ten wykorzystuje gradienty obliczane na podstawie próbek treningowych do aktualizacji wag modelu w celu minimalizacji funkcji straty. Momentum jest używane do przyspieszenia procesu uczenia poprzez uwzględnienie poprzednich wag. Wygenerowany wykres pozwala śledzić postęp treningu, monitorując zmiany w funkcji straty.

- Importowanie potrzebnych bibliotek:

Importowane są biblioteki numpy, matplotlib.pyplot, copy oraz time, które są potrzebne do manipulacji danymi, tworzenia wykresów, tworzenia głębokich kopii obiektów i monitorowania czasu wykonania.

- Inicjalizacja parametrów:

Ustala wartości początkowe dla różnych parametrów używanych w kodzie, takich jak ziarno losowości dla powtarzalności, współczynnik regularyzacji, współczynnik uczenia, liczba iteracji i parametr momentum.

```
np.random.seed(1)
lmd = 0.0001
lr = 0.01
num_iter = 200
rho = 0.90
```

- lmd - współczynnik regularyzacji,
- lr - współczynnik uczenia,
- num_iter - liczba iteracji,
- rho - parametr momentum.

- Wczytanie danych:

Wczytuje zbiór danych treningowych i etykiety z plików .npy.

```
trainX = np.load('train_x_test.npy')
trainY = np.load('train_y_test.npy')
```

- Inicjalizacja wag modelu:

Inicjalizuje wagi modelu i tworzy ich kopię.

```
num_features = trainX.shape[1]
W = np.random.normal(loc=0.0, scale=1.0, size=[trainX.shape[1], 1])
W_old = copy.deepcopy(W)
```

- Inicjalizacja listy do przechowywania strat i licznika iteracji:

Tworzy listę do przechowywania strat dla każdej iteracji i inicjalizuje licznik iteracji.

```
train_losses_sgdn = []
i = 0
```

- Główna pętla treningowa:

Rozpoczyna główną pętlę trenowania, która trwa przez określoną liczbę iteracji. Dla każdej iteracji inicjalizuje sumę strat i licznik próbek. W każdej iteracji oblicza gradient, aktualizuje wagi, oblicza stratę dla bieżącej próbki, dodaje do sumy strat i zwiększa licznik próbek. Oblicza składnik momentum i wykładnik dla aktualizacji SGD-NAG. Oblicza wykładnik do obliczenia straty i gradient do aktualizacji wag. Aktualizuje stare wagi i obecne wagi modelu. Po przejściu przez wszystkie próbki, oblicza średnią stratę dla iteracji, dodaje do listy strat i drukuje wyniki.

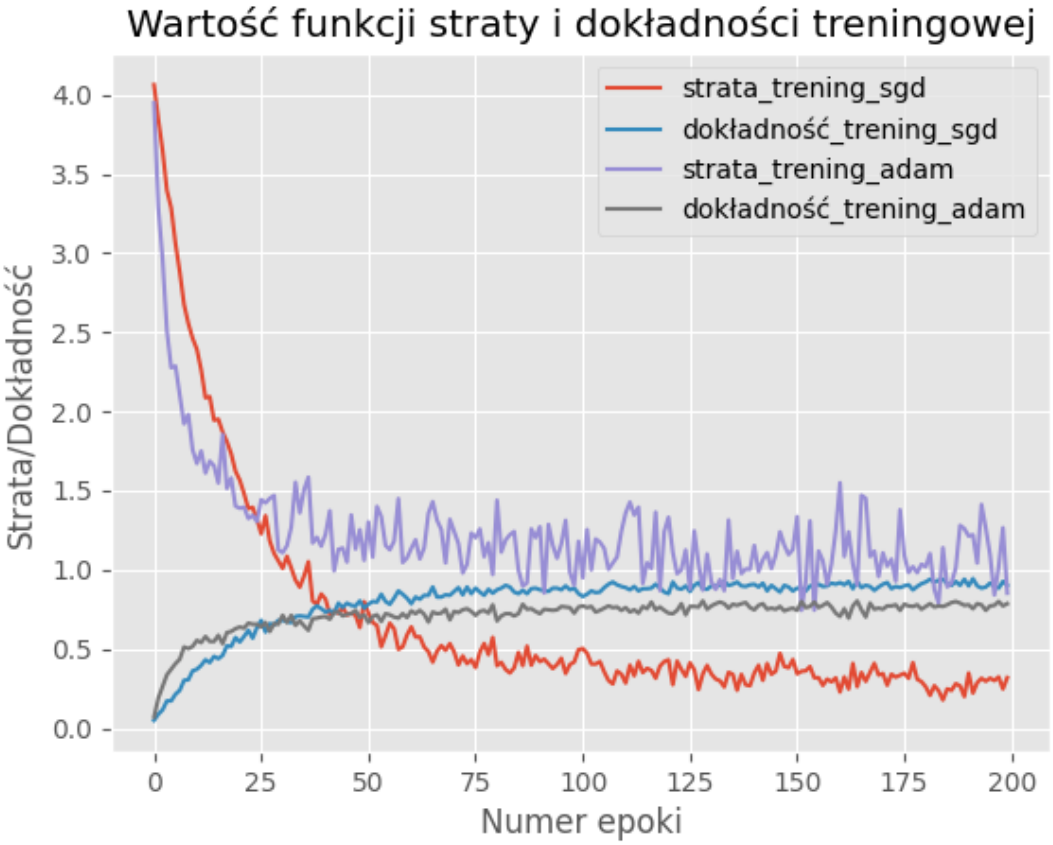
```
for iter in range(num_iter):
    train_loss_sum = 0
    batch_count = 0

    for x, y in zip(trainX, trainY):
        i += 1
        lr = 0.01 / i

        momentum = (1 + rho) * W - rho * W_old
        n_exp = np.exp(-y * np.matmul(x, momentum))

        exp = np.exp(-y * np.matmul(x, W))
        train_grad = -(np.expand_dims(x, axis=1).dot(np.expand_dims(y * n_exp / (1 + n_exp), axis=0))) / \
                     trainX.shape[0] + lmd * momentum

        W_old = copy.deepcopy(W)
        W = momentum - lr * train_grad

        train_loss = np.mean(np.log(1 + exp), axis=0) + (lmd * (np.sum(W ** 2)) / 2)
        train_loss_sum += train_loss
        batch_count += 1

    predictions = np.sign(np.matmul(trainX, W))
    accuracy = np.mean(predictions == trainY)
    if accuracy == 0:
        accuracy = np.finfo(float).eps
    train_accuracies_sgdn.append(accuracy)

    train_losses_sgdn.append(train_loss_sum / batch_count)
    print(iter, train_loss_sum / batch_count)
```

- Tworzenie wykresa

Na końcu tworzy i wyświetla wykres średniej straty na iterację i dokładności treningowej.

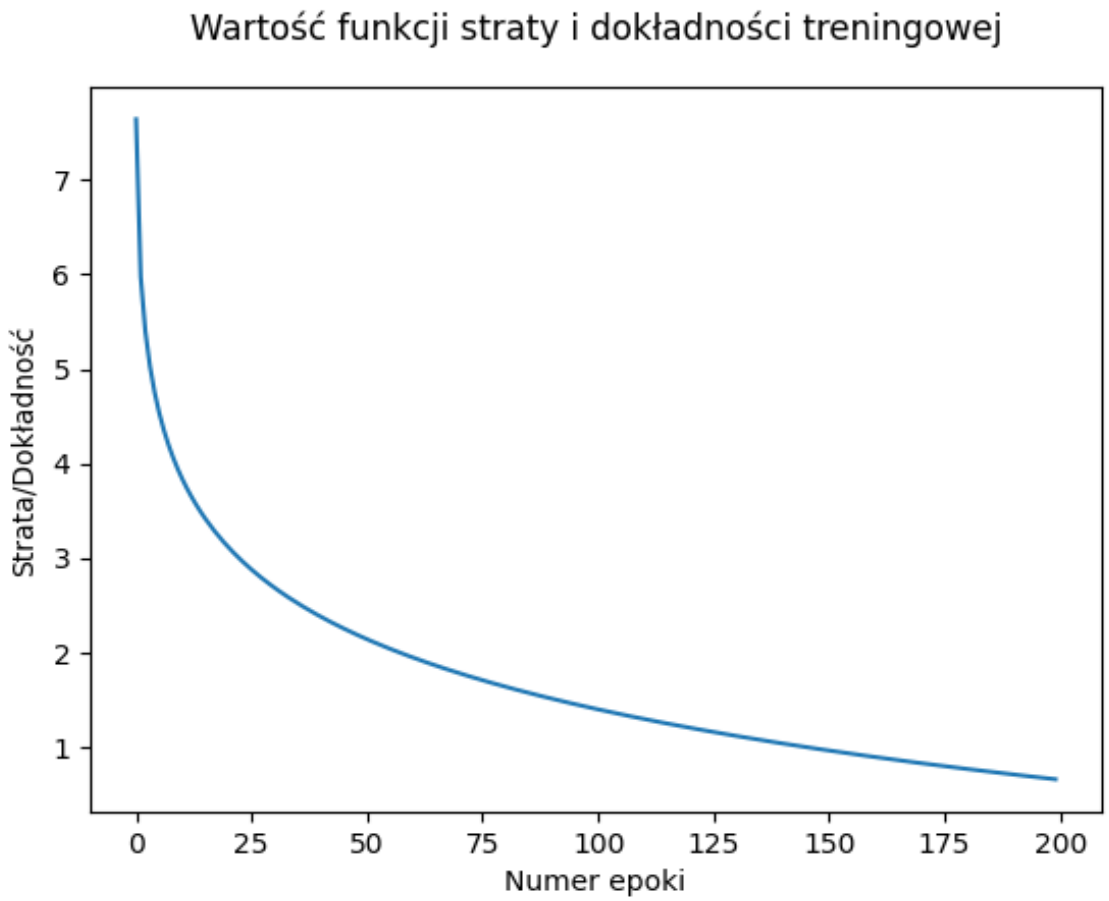
Porównanie metod optymalizacji sieci neuronowych:
Na poniższym wykresie można zauważyć, że metoda stochastycznego spadku gradientowego z przyspieszonym gradientem Nesterov osiąga lepsze rezultaty niż metoda ADAM. Można zauważyć mniejsze wartości funkcji straty oraz większą dokładność w przypadku SGD w porównaniu do ADAM.

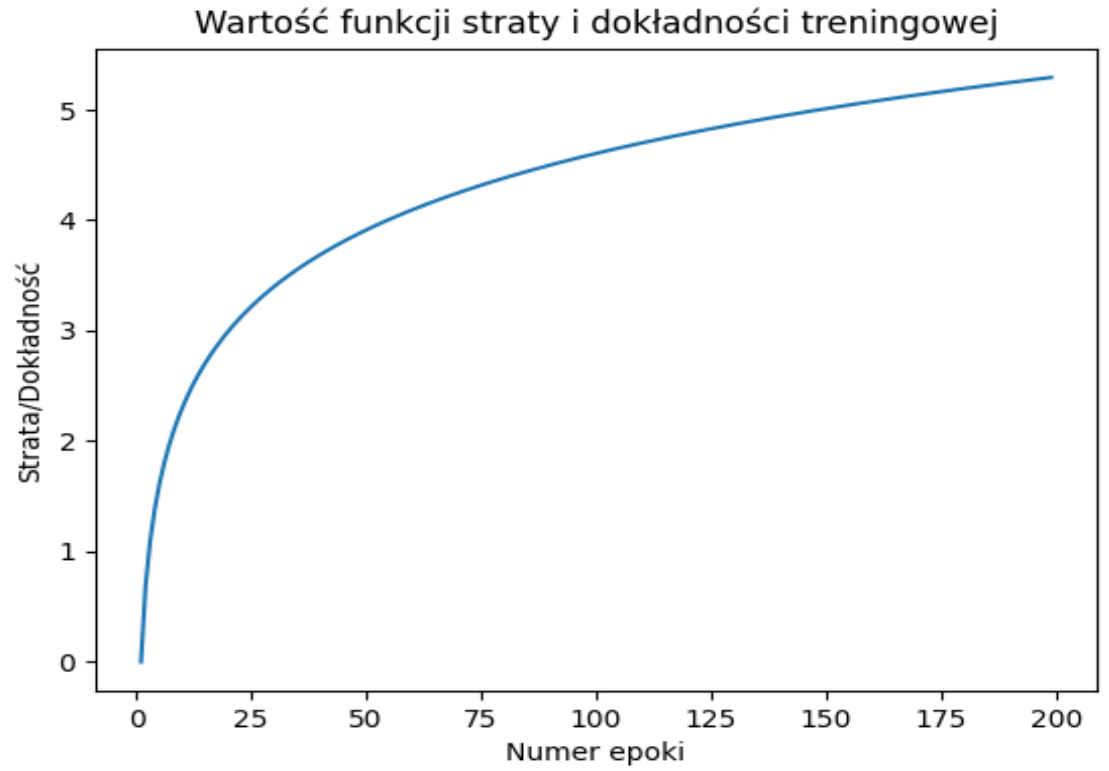
Wyniki na wykresie sugerują, że metoda SGD z przyspieszonym gradientem Nesterov ma mniejsze straty i osiąga wyższą dokładność w porównaniu do metody ADAM. Warto zaznaczyć, że SGD działa dobrze w przypadku dużych zbiorów danych i problemów, w których dane są stosunkowo jednorodne, jednak jest bardziej podatne na oscylacje i wolniejsze zbieganie w porównaniu do bardziej zaawansowanych metod.



Na podstawie analizy można zauważyć, że metoda SGD (stochastyczny spadek gradientu) działa szybciej niż metoda ADAM (Adaptive Moment Estimation).

```
Czas treningu SGD:  27.775918245315555
Czas treningu ADAM:  28.90882182121277
```





Na przedstawionym wykresie widoczne jest, że nasza autorska implementacja SGD-NAG (Stochastic Gradient Descent with Nesterov Accelerated Gradient) daje zadowalające wyniki. Mimo że osiągnięte straty są nieco wyższe, a dokładność nieco niższa w porównaniu do standardowej metody dostępnej w bibliotece, nasza implementacja nadal przynosi

akceptowalne rezultaty. To świadczy o funkcjonalności naszego własnego rozwiązania i możliwości jego wykorzystania w praktyce.

Interfejs chatbota

Chatbot posiada interfejs graficzny, który jest realizowany za pomocą biblioteki tkinter. Składa się on z okna, pola tekstowego do wyświetlania wiadomości, pola tekstowego do wprowadzania wiadomości i przycisku do wysyłania wiadomości. Po wysłaniu wiadomości chatbot generuje odpowiedź i wyświetla ją w polu tekstowym wyjścia.