**ZADANIA Z SIECI NEURONOWYCH 26.11.2023**

**Notacja:**

**SSN/ANN – Sztuczna Sieć Neuronowa**

**CNN – Konwolucyjna Sieć Neuronowa**

**RNN – Rekursywna Sieć Neuronowa**

1. **Propozycja paru prostych ćwiczeń z Pythonem i paczkami do ML.**

**PYTANIA PROBLEMOWE**

1. Z czego wynika potrzeba rozdzielenia zbioru danych na dane uczące i testowe?
2. Załóżmy, że mamy dwie różne sieci neuronowe, uczone niezależnie od siebie na tym samym zbiorze uczącym. Załóżmy też, że rozkład błędu obu sieci na zbiorze testowym jest rozkładem normalnym o zerowej wartości oczekiwanej i standardowych odchyleniach odpowiednio: �1s1 i �2s2. Jaki jest rozkład na zbiorze testowym wartości ½ (y1+y2) gdzie y1, y2 oznaczają wyjścia obu sieci? Jak można wykorzystać ten wynik do poprawy jakości aproksymacji?
3. Załóżmy, że mamy użyć sieci neuronowej do prognozowania przyszłej wartości pewnego procesu zmiennego w czasie, charakteryzującego się tym, że jego przyszłe wartości zależą od przeszłych zgodnie z równaniem:



gdzie t oznacza czas, f jest nieznaną funkcją, zaś h stałą, określającą najdalszą zależność między przeszłością a przyszłością (taki proces jest przykładem tzw. szeregu czasowego). Zaproponować sposób użycia sieci neuronowej do wykonania prognozy. Jak stworzyć zbiór trenujący dla sieci?

1. Czym skutkuje obecność w zbiorze trenującym elementów powtarzających się?
2. Funkcja błędu minimalizowana w czasie uczenia sieci neuronowej ma minima lokalne i punkty siodłowe (w których gradient zeruje się), a także obszary płaskie o bardzo małych wartościach modułu gradientu. Z czego wynikają te zjawiska? Dla jakich wartości wag da się je zaobserwować?
3. Pytanie rekrutacyjne – random.
4. Pytanie rekrutacyjne – random.
5. Pytanie rekrutacyjne – random.
6. Pytanie rekrutacyjne – random.
7. Pytanie rekrutacyjne – random.

**KODOWANIE**

**Uwagi wstępne.**

**Zadania z kodowania traktujemy jak problemy, nad którymi pracujemy na bieżąco a które konsultujemy ze mną. Jest czas na rozwiązanie każdego zadania (90 minut), które stopniowo ujawniam. Nie stresujemy się – nie daję kar za brak rozwiązania. Daję punkty za dobre pomysły. Gdy mamy pytanie, piszemy je do mnie na czacie. Na koniec każdego zadania – rozwiązanie.**

**Rozwiązanie to nie tylko kod samej sieci, ale również przygotowanie danych, test działania itd., komentarz i analiza krytyczna (proszę p. niżej). W razie wątpliwości proszę mnie pytać.**

**Zadania dotyczą wyłącznie sieci neuronowych. Mile widziane, pożądane i punktowane porównanie rozwiązania z (właściwą – jeśli możliwa) metodą uczenia maszynowego poznaną podczas innych zajęć.**

Z1. Najpierw problem klasyfikacyjny. Dla z góry danej trzybitowej dodatniej liczby binarnej (bez znaku) odpowiedzieć przy pomocy sieci neuronowej na pytanie, czy jest ona parzysta.

Obraz zawierający tekst, zrzut ekranu, Czcionka, linia

Opis wygenerowany automatycznie

**Model.**

**Obraz zawierający diagram, linia, Czcionka, Plan

Opis wygenerowany automatycznie**

**Najpierw neurony z warstwy wejściowej są inicjalizowane bitami wejściowej liczby. Następnie wartość każdego neuronu z warstwy wejściowej mnożona jest przez odpowiednią wagę i jest przesyłana do neuronu U1. Neuron U1 sumuje wszystkie trzy wartości. Rezultat wyliczony w neuronie U1 należy jeszcze zinterpretować. O tej wartości można myśleć, jako o pewnej mierze rozrzutu. Przykładowo, jeśli w neuronie U1 dostaniemy liczbę 332482, to nasza sieć neuronowa twierdzi, że z dużym prawdopodobieństwem poprawnym rezultatem dla danych trzech bitów jest 1. Jeśli natomiast neuron U1 wyliczył liczbę -54387, nasza sieć przewiduje, że poprawny wynik to 0. Jeśli natomiast neuron U1 wyliczył wartość 0, nasza sieć neuronowa nie ma bladego pojęcia, jaki wynik jest poprawny.**

**Załóżmy, że wykonujemy jedną iterację procesu trenowania dla pary (110, 1) - pierwsza współrzędna w krotce to dane wejściowe, druga to oczekiwany rezultat. Oznaczmy przez R wartość wyliczoną w neuronie U2 dla opisanych wyżej danych wejściowych i dla wag, jakie w czasie propagacji były przypisane synapsom wychodzącym z neuronów z warstwy wejścia. Propagację wsteczną zaczniemy od wyliczenia tego, jak bardzo wartość wyliczona podczas propagacji różni się od oczekiwanego wyniku. Następnie - w zależności od otrzymanego błędu - należy poprawić wagi synaps. W metodzie użytej w tym przykładzie optymalizacja pojedynczej wagi odbywa się następująco.**

***error = R - expected\_result***

***weight = weight + expected\_result \* error \* d\_sigmoid(R)***

**gdzie d\_sigmoid(R) jest pochodną funkcji sigmoid w punkcie x=R. Orientacyjnie, funkcja mierząca błąd jest funkcją wypukłą, zatem można ją minimalizować schodząc "wzdłuż jej gradientu" - czyli w kierunku jej globalnego minimum.**

**W konstruktorze klasy SimpleNeuralNetwork najpierw inicjalizujemy generator liczb losowych, a następnie określamy początkowe wartości wag liczbami losowymi. Początkowe wagi są zapisane jako współrzędne wektora kolumnowego. Funkcja propagation jest odpowiedzialna za wykonanie procesu propagacji. Dane wejściowe (trzy bity liczby binarnej) są przekazywane do tej funkcji w postaci wektora o trzech współrzędnych. Następnie liczymy iloczyn skalarny wektora wag oraz wektora danych wejściowych, zatem jest to wartość, którą liczy neuron U1. Przekazując tę wartość do funkcji sigmoid otrzymujemy rezultat obliczeń z neuronu U2. Funkcja backward\_propagation implementuje proces propagacji wstecznej. Na początku obliczany jest błąd względem oczekiwanego rezultatu. Następnie modyfikowane są wagi synaps. Instrukcja wykonuje opisane wcześniej obliczenia na każdej współrzędnej wektora wag. Funkcja np.dot odpowiada za mnożenie macierzy. Wyrażenie train\_input. T to po prostu operacja transpozycji wektora (macierzy) train\_input.**

**Szkielet rozwiązania:**

***import numpy as np***

***class SimpleNeuralNetwork:***

***"""***

***Simple NN that checks if a given binary representation of a positive number is even***

***"""***

***def \_\_init\_\_(self):***

***np.random.seed(1)***

***self.weights = 2 \* np.random.random((3, 1)) - 1***

***def sigmoid(self, x):***

***"""***

***Sigmmoid function - smooth function that maps any number to a number from 0 to 1***

***"""***

***return ????***

***def d\_sigmoid(self, x):***

***"""***

***Derivative of sigmoid function***

***"""***

***return ???***

***def train(self, train\_input, train\_output, train\_iters):***

***for \_ in range(train\_iters):***

***propagation\_result = ???***

***self.backward\_propagation(***

***propagation\_result, train\_input, train\_output)***

***def propagation(self, inputs):***

***"""***

***Propagation process***

***"""***

***return self.sigmoid(???)***

***def backward\_propagation(self, propagation\_result, train\_input, train\_output):***

***"""***

***Backward propagation process***

***"""***

***error = ???***

***self.weights += np.dot(***

***train\_input.T, ???***

**)**

**Uwaga. Trzeba jeszcze zrobić testy działania sieci!**

Z2. Czas na konkretny problem regresyjny. Dane wyglądają tak:

Obraz zawierający tekst, zrzut ekranu, Czcionka, numer

Opis wygenerowany automatycznie

Problem polega na predykcji ceny. Do tworzenia modelu nie używamy oczywiście pełnych danych. Niektóre dane są losowo wybierane i odkładane na bok w celu sprawdzenia, jak dobry jest model. Jest to znane podejście jako dane testowe, a pozostałe dane są nazywane danymi treningowymi, na podstawie których jest zbudowany model. Zazwyczaj 70% danych jest używanych jako dane treningowe, a pozostałe 30% jest używane jako dane testowe.

Model. Obraz zawierający diagram, linia, szkic, tekst

Opis wygenerowany automatycznie Na tym samym etapie standaryzujemy również dane. Jest to ważne dla sieci neuronowych, ponieważ poprawia szybkość trenowania modelu i pomaga znaleźć globalne minima. Wykorzystamy moduł "Sequential" z biblioteki Keras służy do tworzenia sekwencji warstw SSN ułożonych jedna po drugiej. Każda warstwa jest definiowana za pomocą modułu "Dense". Keras, w którym określamy, ile neuronów by się tam znajdowało, jaka technika zostałaby użyta do inicjalizacji wag w sieci. jaka będzie funkcja aktywacji dla każdego neuronu w tej warstwie itp. Mamy też hiperparametry:

units=5: Oznacza to, że tworzymy warstwę z pięcioma neuronami. Każdy z tych pięciu neuronów będzie odbierał wartości danych wejściowych, na przykład wartości "Wiek" zostaną przekazane do wszystkich pięciu neuronów, podobnie jak wszystkie inne kolumny.

input\_dim=7: Oznacza to, że w danych wejściowych znajduje się siedem predyktorów, które są oczekiwane przez pierwszą warstwę. Jeśli widzisz drugą gęstą warstwę, nie określamy tej wartości, ponieważ model sekwencyjny przekazuje tę informację dalej do kolejnych warstw.

kernel\_initializer='normal': Kiedy neurony rozpoczynają obliczenia, jakiś algorytm musi zdecydować o wartości dla każdej wagi. Ten parametr określa, że. Możemy wybrać dla niego różne wartości, takie jak "normalny" lub "glorot\_uniform".

activation='relu': Określa funkcję aktywacji dla obliczeń wewnątrz każdego neuronu. Możesz wybrać wartości takie jak "relu", "tanh", "sigmoid" itp.

batch\_size=20: Określa, ile wierszy zostanie przekazanych do sieci za jednym razem, po czym rozpocznie się obliczanie SSE, a sieć neuronowa zacznie dostosowywać swoje wagi na podstawie błędów.

Gdy wszystkie wiersze są przekazywane w partiach po 20 wierszy, jak określono w tym parametrze, nazywamy to 1-epoką. Lub jeden pełny cykl danych. Jest to również znane jako zejście gradientu mini-batch. Mała wartość batch\_size spowoduje, że SSN będzie patrzeć na dane powoli, na przykład 2 wiersze na raz lub 4 wiersze na raz, co może prowadzić do nadmiernego dopasowania, w porównaniu z dużą wartością, taką jak 20 lub 50 wierszy na raz, co sprawi, że SSN będzie szybko patrzeć na dane, co może prowadzić do niedopasowania. W związku z tym należy wybrać odpowiednią wartość przy użyciu dostrajania hiperparametrów.

Epoki=50: Ta sama czynność dostosowywania wag jest kontynuowana 50 razy, zgodnie z tym parametrem. Mówiąc prościej, SSN patrzy na pełne dane treningowe 50 razy i dostosowuje swoje wagi.

Znalezienie najlepszych wartości dla batch\_size i epoki jest bardzo ważne, ponieważ ma bezpośredni wpływ na wydajność modelu. Złe wartości mogą prowadzić do nadmiernego lub niedostatecznego dopasowania. Nie ma reguły, która mogłaby pomóc w podjęciu decyzji o liczbie warstw/liczbie neuronów itp. przy pierwszym spojrzeniu na dane. Musimy wypróbować różne parametry i wybrać kombinację, która zapewnia jak najwyższą dokładność. Pamiętajmy tylko, że im większa sieć, tym bardziej obciąża ona obliczeniowo, dlatego jej uruchomienie zajmie więcej czasu. Dlatego zawsze należy znaleźć najlepszą dokładność przy minimalnej liczbie warstw/neuronów.

Pierwsze podejście do szukania parametrów to znajdowanie najlepszego zestawu parametrów za pomocą ręcznego wyszukiwania w ich siatce. Jest to proste podejście oparte na pętli. Można to łatwo edytować i dostosować do większej liczby hiperparametrów, po prostu dodając kolejną zagnieżdżoną pętlę. Korzystając z najlepszego zestawu parametrów znalezionych powyżej, ponownie trenuj model i przewiduj ceny na danych testowych. Korzystając z końcowego wytrenowanego modelu, teraz generujemy błąd przewidywania dla każdego wiersza w danych testowych jako bezwzględny błąd procentowy. Przyjęcie średniej dla wszystkich wierszy jest znane jako średni bezwzględny błąd procentowy (MAPE). Dokładność jest obliczana jako 100-MAPE.

**Szkielet kodu:**

***import pandas as pd***

***import numpy as np***

***# To remove the scientific notation from numpy arrays***

***# Separate Target Variable and Predictor Variables***

***X=CarPricesDataNumeric[Predictors].values***

***y=CarPricesDataNumeric[TargetVariable].values***

***### Sandardization of data ###***

***from sklearn.preprocessing import StandardScaler***

***# Storing the fit object for later reference***

***PredictorScalerFit=PredictorScaler.fit(X)***

***TargetVarScalerFit=TargetVarScaler.fit(y)***

***# Generating the standardized values of X and y***

***# Split the data into training and testing set***

***from sklearn.model\_selection import train\_test\_split***

***# Quick sanity check with the shapes of Training and testing datasets***

***# importing the libraries***

***from keras.models import Sequential***

***from keras.layers import Dense***

***# create ANN model***

***model = Sequential()***

***# Defining the Input layer and FIRST hidden layer, both are same!***

***model.add(Dense(units=5, input\_dim=7, kernel\_initializer='normal', activation='relu'))***

***# Defining the Second layer of the model***

***# after the first layer we don't have to specify input\_dim as keras configure it automatically***

***# The output neuron is a single fully connected node***

***# Since we will be predicting a single number***

***# Compiling the model***

***# Fitting the ANN to the Training set***

***# Defining a function to find the best parameters for ANN***

***def FunctionFindBestParams(X\_train, y\_train, X\_test, y\_test):***

***# Defining the list of hyper parameters to try***

***batch\_size\_list=[5, 10, 15, 20]***

***epoch\_list = [5, 10, 50, 100]***

***import pandas as pd***

***SearchResultsData=pd.DataFrame(columns=['TrialNumber', 'Parameters', 'Accuracy'])***

***# initializing the trials***

***TrialNumber=0***

***for batch\_size\_trial in batch\_size\_list:***

***for epochs\_trial in epoch\_list:***

***TrialNumber+=1***

***# create ANN model***

***model = Sequential()***

***# Defining the first layer of the model***

***# Defining the Second layer of the model***

***# The output neuron is a single fully connected node***

***# Since we will be predicting a single number***

***# Compiling the model***

***# Fitting the ANN to the Training set***

***MAPE = np.mean(100 \* (np.abs(y\_test-model.predict(X\_test))/y\_test))***

***# printing the results of the current iteration***

***SearchResultsData=SearchResultsData.append(pd.DataFrame(data=[[TrialNumber, str(batch\_size\_trial)+'-'+str(epochs\_trial), 100-MAPE]],***

***columns=['TrialNumber', 'Parameters', 'Accuracy'] ))***

***return(SearchResultsData)***

***######################################################***

***# Calling the function***

***ResultsData=FunctionFindBestParams(X\_train, y\_train, X\_test, y\_test)***

***# Function to generate Deep ANN model***

***def make\_regression\_ann(Optimizer\_trial):***

***from keras.models import Sequential***

***from keras.layers import Dense***

***model = Sequential()***

***return model***

***# Fitting the ANN to the Training set***

***model.fit(X\_train, y\_train ,batch\_size = 15, epochs = 5, verbose=0)***

***# Generating Predictions on testing data***

***Predictions=model.predict(X\_test)***

***# Scaling the predicted Price data back to original price scale***

***Predictions=TargetVarScalerFit.inverse\_transform(Predictions)***

***# Scaling the y\_test Price data back to original price scale***

***y\_test\_orig=TargetVarScalerFit.inverse\_transform(y\_test)***

***print("########## Total Time Taken: ", round((EndTime-StartTime)/60), 'Minutes')***

***print('### Printing Best parameters ###')***

***grid\_search.best\_params\_***

***# Scaling the test data back to original scale***

***Test\_Data=PredictorScalerFit.inverse\_transform(X\_test)***

***TestingData=pd.DataFrame(data=Test\_Data, columns=Predictors)***

***TestingData['Price']=y\_test\_orig***

***TestingData['PredictedPrice']=Predictions***

***TestingData.head()***

***# Computing the absolute percent error***

***APE=100\*(abs(TestingData['Price']-TestingData['PredictedPrice'])/TestingData['Price'])***

***TestingData['APE']=APE***

***print('The Accuracy of ANN model is:', 100-np.mean(APE))***

***TestingData.head()***

***###########################################***

***from sklearn.model\_selection import GridSearchCV***

***from keras.wrappers.scikit\_learn import KerasRegressor***

***# Listing all the parameters to try***

***Parameter\_Trials={'batch\_size':[10,20,30],***

***'epochs':[10,20],***

***'Optimizer\_trial':['adam', 'rmsprop']***

***}***

***# Creating the regression ANN model***

***RegModel=KerasRegressor(make\_regression\_ann, verbose=0)***

***###########################################***

***from sklearn.metrics import make\_scorer***

***# Defining a custom function to calculate accuracy***

***def Accuracy\_Score(orig,pred):***

***MAPE = np.mean(100 \* (np.abs(orig-pred)/orig))***

***print('#'\*70,'Accuracy:', 100-MAPE)***

***return(100-MAPE)***

***custom\_Scoring=make\_scorer(Accuracy\_Score, greater\_is\_better=True)***

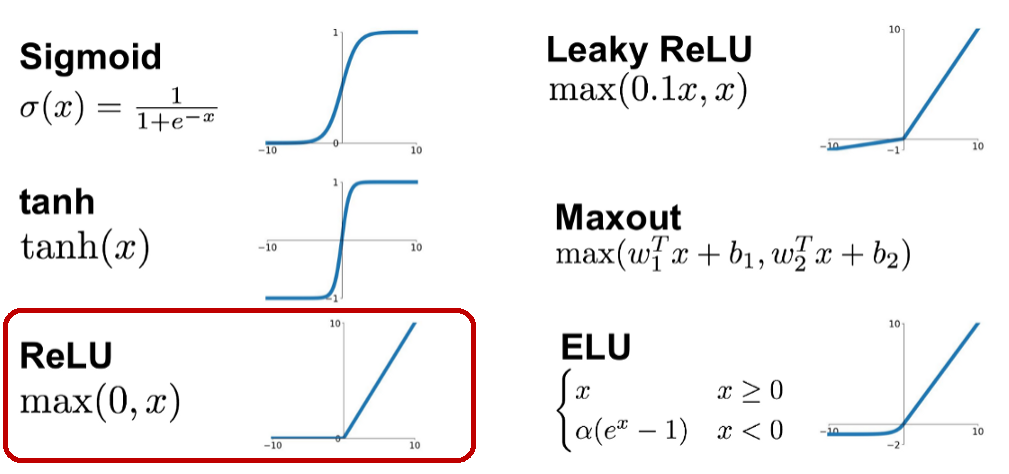
Z3. Zbiór danych Fashion MNIST został przygotowany przez zespół Zalando. Składa się on z obrazów oraz przypisanego do nich opisu garderoby. Zbiór treningowy ma 60.000 obserwacji a testowy 10.000. Naszą cechą jest obraz, który jest tabelą w NumPy w rozmiarze 28 na 28 pikseli. Wartości w tej macierzy przyjmują wartość od 0 do 255 co odpowiada wartościom kolorów w notacji RGB (Red Green Blue). Wartość liczbowa piksela ma odpowiednik w skali RGB, np. wartość 0 w notacji RGB (0, 0, 0) to kolor czarny a 255 to kolor biały.

Sieć neuronowa musi mieć poprawnie przygotowane dane. Jedną z ważniejszych zasad jest, że dane wejściowe muszą mieć taki sam rozmiar. Zatem w przypadku zdjęć odpowiednio się je obcina do takiej samej wielkości lub stosuje się inne metody. W przypadku wrzucania danych tekstowych należy również odpowiednio obciąć zbyt długie sekwencje do krótszej długości, a te które były zbyt krótkie uzupełnić np. zerami. W naszym przypadku nie mamy problemu, ponieważ zbiór jest już przygotowany i każda obserwacja ma na wejściu obraz składający się z 28 na 28 pikseli czyli posiada ten sam rozmiar. Inaczej mówiąc każdy piksel jest osobną cechą, czyli tak jakbyśmy mieli 784   
(28 x 28) charakterystyk opisujących nasz ubiór, który chcemy przewidzieć. Parametry naszej sieci neuronowej najczęściej są inicjowane za pomocą niewielkich liczb losowych. Czasami może się okazać, że skonstruowana sieć neuronowa może niezbyt dobrze radzić sobie w przypadkach, gdy wartości cech są znacznie większe. Pamiętajmy też, że wejście do kolejnych węzłów to waga przemnożona przez wartość i nałożona na to funkcja aktywacji. Stąd można wnioskować, że jeśli charakterystyki wejściowe mają podobną skalę wówczas znacznie ułatwimy naszej sieci przygotowanie predykcji. Dlatego dobrą praktyką jest przygotowanie standaryzacji cech. Najprościej wykorzystać wbudowane klasy z biblioteki scikit-learn. Natomiast w naszym przypadku (gdzie mamy wartości z zakresu od 0 do 255) wystarczy podzielić liczby przez 255 i będziemy mieć dane z przedziału jednostkowego.

Naszym zadaniem jest przewidzenie 10 klas. Wobec tego naszym celem jest przewidzenie prawdopodobieństwa dla każdej z tych klas. Istnieją dwa interfejsy API do definiowania modelu   
w Keras:

* sekwencyjny (Sequential model API) – jest prostszy, składa się z kolejnych warstw,
* funkcjonalny ( Functional API) – jest odrobinę bardziej skomplikowany, ale dzięki temu można składać bardziej złożone modele z wieloma wejściami czy wyjściamy, sklejać w locie czy współdzielić warstwami.

Tu wystarczy interfejs sekwencyjny. Zacznijmy od tego, że na wejściu model otrzymuje macierz dla każdej obserwacji 28 x 28. Wobec tego spłaszczmy ją do pojedynczego wektora 1 x 784. Pierwsza warstwa zawsze musi posiadać parametr input\_shape, który mówi o tym jakie dane dostanie sieć neuronowa na początku. Na początku dodajmy tylko jedną warstwę ukrytą składającą się z 128 neuronów i funkcji aktywacji ReLU (ang. rectified linear unit). Jest to najbardziej znana i najczęściej stosowana funkcja w przypadku warstw ukrytych.



W naszym przypadku w pierwszym podejściu definiujemy tylko jedną warstwę ukrytą. Oczywiście tak jak w przypadku liczby neuronów im więcej warstw ukrytych tym więcej nasza sieć neuronowa będzie mogła się nauczyć kosztem wydłużenia czasu lub przeuczeniem. W warstwie gęstej (dense) wszystkie jednostki poprzedniej warstwy są połączone ze wszystkimi w następnej. Ostatnia warstwa, nazywana warstwą wyjściową, powinna mieć zdefiniowaną liczbę neuronów zależną od tego, co przewidujemy. W naszym przypadku chcemy przewidzieć 10 klas, stąd przyjmiemy liczbę neuronów równą 10.

Oprócz tego należy zdefiniować strukturę funkcji aktywacji naszej warstwy wyjściowej. To jaką przyjmujemy funkcję aktywacji w dużym znaczeniu zależy od tego co chcemy prognozować. Przykładowo dla:

* klasyfikacji binarnej najlepiej przyjąć funkcję aktywacji o kształcie „S”,
* klasyfikacji wielowymiarowej najczęściej stosowana jest znormalizowana funkcja wykładnicza softmax,

W naszym przykładzie jest 10 klas więc użyjemy softmax. Dzięki temu otrzymamy na wyjściu tablicę składającą się z 10 wartości dla naszej obserwacji sumujących się do 1. Zatem te 10 wartości można traktować jako prawdopodobieństwa, że nasza obserwacja należy do jednej z 10 klas.

W tym kroku są jeszcze przed nami postawione dwa ważne zadania. Pierwsze z nich to określenie funkcji straty, czyli funkcji która będzie sprawdzać jak nasza prognoza z sieci myli się od prawdziwej wartości. W tym miejscu najistotniejszy jest rodzaj problemu, który mamy. Najczęściej polecane są tutaj poniższe propozycje:

* dla klasyfikacji binarnej użycie binarnej entropii krzyżowej (binary\_crossentropy),
* dla klasyfikacji wielowymiarowej użycie kategoryzującej entropii krzyżowej (categorical \_crossentropy),
* w przypadku regresji błąd średniokwadratowy (mse).

Drugim zadaniem jest określenie algorytmu optymalizacji. Jego zadaniem jest poprawne pokierowanie funkcji straty do znalezienia jak najmniejszego błędu. Tutaj również jest wiele algorytmów do dyspozycji. Jedne z nich to:

* propagacja RMS,
* propagacja ADAM (ang. adaptive moment estimation),
* stochastyczny spadek wzdłuż gradientu i metoda pędu (AGD).

Z4. Rozpatrzymy przykład sieci, której zadaniem będzie klasyfikacja pary liczb do dwóch grup. Albo grupy, której wynik operacji XOR da 0 lub grupy, której wynik operacji XOR da 1. XOR polega ona na przyjęciu: dwóch liczb (zazwyczaj są to liczby naturalne 0 lub 1) oraz tego, że oczekiwany wynik jest równy zeru, gdy dwie liczby są takie same. Dla par (0,0) i (1,1) otrzymujemy na wyjściu 0, natomiast dla par różniących się (0,1) lub (1,0) otrzymujemy na wyjściu 1. Problem pojawia się w momencie, gdy założymy, że nasze liczby nie są liczbami całkowitymi, to znaczy jak mamy interpretować parę liczb (0.1,1.3)? Wyobraźmy sobie, że musimy dokonać operacji XOR dwóch wejść, które podają napięcie z zakresu od 0V do 1.3V. To napięcie prawie zawsze nie będzie idealnie równe liczbie całkowitej. Dla takiego układu musimy stworzyć model, który będzie potrafił sklasyfikować (stąd nazwa sieć klasyfikacyjna lub klasyfikator) daną parę liczb do dwóch grup – albo do grupy par liczb, dających w wyniku operacji XOR 0 lub do grupy par liczb, które w wyniku operacji XOR dadzą 1. Będziemy realizować cel wygenerowania losowych danych, na których dokonamy późniejszego treningu (uczenia) naszej sieci:

***import torch***

***import torch.utils.data as data***

***class XORDataCreator(data.Dataset):***

***""" XORDataCreator jest klasą, która generuje losowe dane treningowe dla głębokiej sieci klasyfikacyjnej.***

***Klasyfikator, który korzysta z generowanych danych powinien otrzymać parę liczb zmiennoprzecinkowych obarczonych szumem oraz oznaczenie pary liczb jakiego wyniku alternatywy rozłącznej (operacja XOR) spodziewamy się po tej parze. Na tej podstawie sieć ma nauczyć się rozpoznawać wyniki z możliwie jak najwyższą poprawnością nieznanej dotąd pary liczb zmiennoprzecinkowych."""***

***# Definicja konstruktora (metoda inicjalizacyjna klasy)***

***# data\_size: ilość danych losowych, która ma zostać wygenerowana przez obiekt tej klasy***

***# noise\_std\_deviation: odchylenie standardowe szumu Gaussowskiego (zaszumienie danych ma na celu przygotować trening sieci na mniej standardowe dane)***

***def \_\_init\_\_(self, data\_size, noise\_std\_deviation) -> None:***

***super().\_\_init\_\_()***

***self.data\_size = data\_size***

***self.noise\_std\_deviation = noise\_std\_deviation***

***self.generate\_random\_xor\_data()***

***def generate\_random\_xor\_data(self):***

***# Generacja losowych par liczb (x,y) z przedziału <0,2>***

***xor\_data = torch.randint(low=0, high=2, size=(self.data\_size, 2), dtype=torch.float32)***

***# Generacja szumu dla danych (losowy tensor o tych samych wymiarach co nasze dane wygenerowane linię wyżej z zakresu <0,1> pomnożony przez odchylenie standardowe)***

***xor\_data\_noise = ???***

***# Labelling danych oznacza nadanie każdemu elementowi z tablicy danych oczekiwanego wyniku***

***# W tym przypadku aby określić oczekiwany wynik dokonujemy operacji XOR (operator ^).***

***labels = ???***

***# Zapisujemy wygenerowane dane do pól klasy prezentujących zbiór danych treningowych oraz nadanym im oczekiwanych wyników (labels)***

***xor\_data\_with\_noise = ???***

***labels\_converted\_to\_numbers = labels # Lista 'labels' zawiera jednoelementowe tensory - poprzez metodę item() wyłuskujemy wartość***

***self.data\_inputs = xor\_data\_with\_noise***

***self.data\_labels = [label.to(torch.long) for label in labels]***

***def \_\_len\_\_(self):***

***return ???***

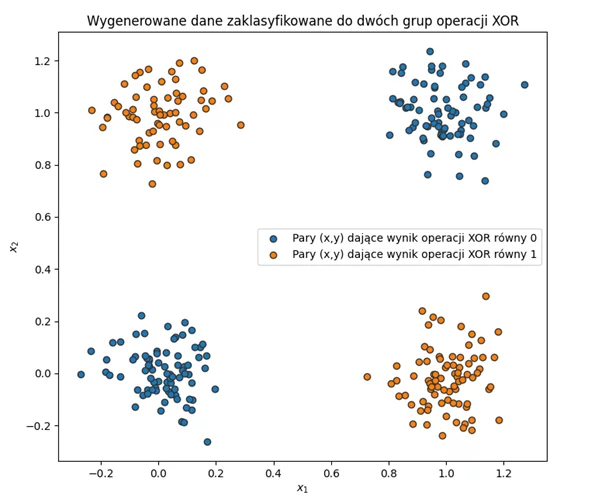
***def \_\_getitem\_\_(self, index):***

***return ???***

***# Użycie wyżej zdefiniowanej klasy do generacji 300 elementowego tensora danych***

***xor\_dataset = ???***

Po uruchomieniu naszego program zmienna xor\_dataset powinna zawierać tensor (a dokładniej parę określaną jako tuple) dwóch elementów – jeden z nich to kolejny tensor, który zawiera zbiór 300 elementów par losowo wygenerowanych liczb, a drugi element to oczekiwany wynik operacji XOR pary liczb z pierwszego tensora. Oczekiwany wynik jest podawany w sieci, aby ta mogła na bieżąco „dostrajać” wagi w tym przypadku dwóch wejść (na wejściu mamy parę liczb) tak, aby wyjście z jak największym prawdopodobieństwem pokrywało się z tym, czego oczekujemy na podstawie wygenerowanych danych.



Jak widać, pary liczb, które są różne (zaokrąglają się do różnych liczb całkowitych) i wedle operacji XOR dają wynik 1 (pomarańczowe punkty), znajdują się w lewej górnej oraz prawej dolnej części wykresu, przeciwnie do par, które zaokrąglają się do tych samych wartości. Teraz zajmiemy się próbą zdefiniowania architektury oraz działania naszego modelu głębokiej sieci neuronowej tak, aby ta mogła dla dowolnego punktu danego tej sieci (nawet takiego, którego nie było w danych, na których sieć była uczona) określić, do jakiej grupy należy.

Oczywiście, ten problem jest stosunkowo prosty celem zaprezentowania zwięzłego przykładu, ale każdy problem, który da się opisać liczbowo, może później zostać poddany próbie nauczenia sieci tego, w jaki sposób te dane są powiązane z informacją końcową, która interesuje użytkownika (na przykład wyniki badań krwi z daną jednostką chorobową).

Skupimy się teraz na zbudowaniu modelu sieci neuronowej oraz zdefiniowaniu procesu treningu (uczenia) sieci. Pierwszym szczegółem jest użycie modułu torch.nn – skrót „nn” pochodzi od pojęcia neural network. Jest to moduł, który pozwala nam w łatwy sposób zdefiniować strukturę naszego modelu sieci neuronowej. W naszym przypadku, ponieważ jest to dość prosty problem, jest to struktura składająca się z:

* Warstwy wejściowej (wejściowa para liczb),
* Warstwy zwanej ukrytą (wszystkie warstwy głębokich sieci neuronowych poza warstwą wejściową i wyjściową nazywane są zwyczajowo ukrytymi),
* Warstwą wyjściową, która prezentuje już interesujący nas wynik.

Poniżej zdefiniowana klasa ClassifierModel definiuje model naszej głębokiej sieci klasyfikatora.   
W konstruktorze za argumenty wejściowe przyjmujemy ilość neuronów poszczególnych warstw (ilość komórek przyjmujących dane oraz propagujących je dalej w przetworzonej formie). Skupmy się na wyjaśnieniu zdefiniowanych relacji między warstwami – tutaj widać konkretne odwołanie do modułu nn zawierający wiele modeli klas, które pozwalają nam w prosty i przejrzysty sposób zaprojektować naszą sieć.

***class ClassifierModel(nn.Module):***

***def \_\_init\_\_(self, inputs\_number, hidden\_layer\_neurons\_number, outputs\_number) -> None:***

***super().\_\_init\_\_()***

***self.first\_layer\_to\_hidden\_transformation = nn.Linear(inputs\_number, hidden\_layer\_neurons\_number)***

***self.activation\_function = ???***

***self.hidden\_layer\_to\_output\_transformation = ???***

***def forward(self, input):***

***x = self.first\_layer\_to\_hidden\_transformation(input)***

***x = self.activation\_function(x)***

***x = self.hidden\_layer\_to\_output\_transformation(x)***

***return x***

Idąc kolejno po każdej linii naszej klasy, napotykamy na definicję pierwszej transformacji (na czerwono). Definiujemy tę transformację jako liniową (nn.Linear), czyli taką, która jest standardową definicją wiążącą wejścia, które są ważone (nadawane są im wagi, czyli liczby, przez które są mnożone wejścia, a ustalane są przez sieć w procesie uczenia tak, aby jak najlepiej odzwierciedlać wyuczonym modelem opisaną w danych treningowych przez nas sytuację), a zaprezentowane mogą zostać za pomocą dość prostego wzoru funkcji liniowej. Funkcja liniowa poprzez odpowiedni dobór wag w wektorze W stara się zminimalizować błąd (pomyłkę) modelu sieci na podstawie dostarczonych danych treningowych. Opisywany problem jest dość prosty i da się opisać analitycznie w sposób liniowy, natomiast niestety większość musi zostać opisana za pomocą nieliniowych funkcji. Następną ważną składową naszej sieci jest funkcja aktywacji, która w naszym przypadku jest funkcją równą tangensowi hiperbolicznemu (nn.Tanh()). Wartości naszej funkcji aktywacji wchodzą w zakres pomiędzy -1 a 1. Jest to świetna funkcja, która wprowadza poprzez swoją nieliniowość możliwość rozwiązania nieliniowych problemów. W dodatku jest wycentrowaną funkcją, ponieważ dla argumentu zero również przyjmuje wartość równą zeru — przecina osie dokładnie w punkcie (0,0). Z tego powodu, podczas gdy nasza sieć się uczy i próbuje poznać, jak wrażliwa jest funkcja błędu (pomyłki) na wspomniane wcześniej wagi każdego z wejść (licząc gradient tej funkcji), korzysta z przywileju, iż gradient tangensu hiperbolicznego jest symetryczny względem osi zerowej (y = 0).

To wielka przewaga tej funkcji nad wieloma pozostałymi nieliniowymi funkcjami aktywacji. Jednakże różne klasy problemów do rozwiązania mają swoje empiryczne rekomendacje odnośnie zastosowania konkretnej funkcji aktywacji. W internecie możemy spotkać mnóstwo dobrej jakości poradników w tym zakresie i warto z nich korzystać, ponieważ zazwyczaj są one poparte praktyką.

Ostatnią transformacją wiążącą warstwę ukrytą i warstwę wyjściową jest również relacja liniowa. Funkcja forward zdefiniowana w tej samej klasie ma za zadanie przyjąć wektor wejściowy i „przepuścić” go przez kolejne warstwy, po czym zwrócić tensor wyjściowy. Taki tensor jest dla nas po prostu informacją o tym, jaki wynik operacji XOR otrzymamy. Wynikiem będzie liczba zmiennoprzecinkowa pomiędzy 0 a 1, gdyż wynik operacji XOR to właśnie w idealnym przypadku 0 lub 1 – tutaj sieć o tym nie wie, dlatego z pewnym prawdopodobieństwem przybliża się ku zeru lub jedynce, stąd zmiennoprzecinkowość.

W skrócie, cały proces przygotowania oraz treningu naszej sieci będzie przebiegał w następujących krokach:

*Inicjalizacja klasy odpowiedzialnej za dostarczenie danych treningowych.*

*Pobranie częściowej paczki danych z całościowej kolekcji danych treningowych sieci.*

*Obliczenie przez obecny model sieci wyjścia dla zaprezentowanych danych wejściowych.*

*Obliczenie błędu, czyli różnicy pomiędzy wynikiem oczekiwanym (podanym na przykład przez nas w klasie odpowiedzialnej za dostarczenie danych – będzie to na przykład dla pary liczb (0,1) oczekiwany wynik 1, ponieważ operacja XOR dla tej pary daje właśnie taki wynik) a tym, jaki wynik otrzymaliśmy w punkcie 3.*

*Dokonanie propagacji wstecznej błędu, czyli sprawdzenie na podstawie funkcji gradientu (czyli funkcji prezentującej wrażliwość funkcji błędu na wagi nadane konkretnym wejściom), w którym punkcie jesteśmy (czy zwiększając na przykład wagę danego wejścia zwiększymy, czy zmniejszymy błąd – będziemy zawsze chcieli go zmniejszyć).*

*Aktualizacja wag tak, aby zmniejszyć błąd sieci.*

*Gdy błąd jest na tyle mały, że uznamy go za satysfakcjonujący, zapisujemy model sieci.*

Punkty te będziemy powtarzać cyklicznie aż osiągniemy zadowalającą dokładność przewidywania wyniku przez model sieci.

Przejdźmy zatem do kluczowej części kodu, czyli treningu naszej sieci. W tym celu, posługując się powyższą instrukcją, przeanalizujemy kod, którego zadaniem będzie w ogólności trening naszego modelu sieci.

***import torch***

***import torch.nn as nn***

***from classifier\_model import ClassifierModel***

***from data\_loader import XORDataCreator***

***class Net():***

***def \_\_init\_\_(self) -> None:***

***self.loss\_calculation\_model = None***

***self.optimizer = None***

***self.data\_loader = XORDataCreator(data\_size=100, noise\_std\_deviation=0.1)***

***def train(self, model: ClassifierModel, epochs\_num: int=50):***

***# Sprawdźmy czy możemy wykorzystać GPU poprzez pakiet CUDA celem przyspieszenia obliczeń***

***gpu\_available = torch.cuda.is\_available()***

***# W przypadku możliwości ustawmy 'device' na GPU***

***device = torch.device('cuda') if gpu\_available else torch.device('cpu')***

***# Inicjalizacja***

***self.loss\_calculation\_model = nn.BCEWithLogitsLoss()***

***self.optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)***

***# Załadujmy nasz model do GPU jeśli jest dostępne***

***model.to(device)***

***# Ustawmy model w tryb treningowy (funkcjonalność odziedziczona po klasie nn.Module)***

***model.train()***

***# Pętla treningowa (pobierz paczkę danych treningowych)***

***for current\_epoch in range(epochs\_num):***

***epoch\_loss = 0.0***

***for data\_input, data\_label in self.data\_loader:***

***# Załadujmy nasze dane do GPU jeśli jest w użyciu***

***data\_input = data\_input.to(device)***

***data\_label = data\_label.to(device)***

***# Przeprocedujmy dane wejściowe przez nasz model***

***predicted\_output = ???***

***predicted\_output = ???***

***# Obliczmy wartość funkcji straty (jak bardzo nasz model się pomylił)***

***loss = self.loss\_calculation\_model(predicted\_output, data\_label.float())***

***# Wyzerujmy wartość gradientu na wszelki wypadek***

***???***

***# Dokonajmy propagacji wstecznej błędu***

***???***

***# Zaktualizujmy wagi sieci na podstawie obliczonego gradientu***

***???***

***# Obliczenie średniej pomyłki obecnej iteracji treningowej***

***epoch\_loss += ???***

***# Logowanie istotnych danych odnośnie zakończonej iteracji***

***print('epoch: ', ???)***

***print('average epoch loss: ', ???)***

***def set\_optimizer(self, optimizer) -> None:***

***self.optimizer = ???***

***def set\_loss\_calc\_model(self, loss\_calc\_model) -> None:***

***self.loss\_calculation\_model = ???***

***def get\_training\_data(self) -> XORDataCreator:***

***return ???***

Zazwyczaj trening skomplikowanych modeli sieci neuronowych jest czasochłonny – zatem pozostaje pytanie, co można zrobić, aby nie utracić danych treningowych w losowych przypadkach lub aby móc podzielić ten czasochłonny proces na kilka mniejszych? Oczywiście można zapisać parametry tymczasowego modelu do pliku, a następnie wczytać ostatnie zapisane podczas ponownej sesji treningowej. Aby wczytać oraz zapisać stan bieżący modelu, należy użyć wbudowanych metod pakietu torch.

***from os.path import exists***

***import torch***

***from net import Net***

***from classifier\_model import ClassifierModel***

***\_\_clasifier\_model\_state\_file\_name\_\_ = "classifier\_model.pt"***

***classifier\_model = ClassifierModel(???)***

***# Wczytaj poprzednio zapisany model, jeśli istnieje***

***if exists(\_\_clasifier\_model\_state\_file\_name\_\_):***

***model\_state = torch.load(???)***

***classifier\_model.load\_state\_dict((???)***

***net\_model = ???***

***net\_model.train\_and\_log(model=???)***

***# Zapisz dotychczasowy model***

***model\_state = classifier\_model.state\_dict()***

***torch.save(???)***

***print("Trained model state dictionary saved to ", ???)***

Z5. Na podstawie dotychczasowych przykładów samodzielnie napisać model klasyfikacyjny cukrzyków dla zbioru danych Pima.