Sprawozdanie - SI

Lab 5

Wiktor Sadowy

Cel zadania

W ramach zadania mieliśmy:

a) Przygotować zbiór uczący i walidacyjny korzystając z dołączonej do listy procedury ekstrakcji cech
b) Przetestować działanie podstawowego modelu MLP
c) Zbadać wpływ tempa uczenia na osiągane wyniki
d) Zbadać wpływ rozmiaru modelu MLP na osiągane wyniki
e) Przetestować najlepszy model w praktyce
f) Wykonać badanie dowolnego innego parametru (w naszym przypadku regularyzacji) – zadanie dodatkowe

Wszystkie zadania miały zostać wykonane z użyciem datasetu Jester.

Opis teoretyczny

W ramach zadania zajmujemy się sieciami neuronowymi czyli szeroką klasą systemów służącą do przetwarzania informacji, którą wspólną cechą jest struktura składająca się z neuronów, zorganizowanych w warstwy połączonych ze sobą połączeniami o przypisanych wagach.

Jednym z przykładów sieci neuronowej jest Multilayer Perceptron (MLP). Jest to jednokierunkowa sieć neuronowa o jednej warstwie wejściowe, jednej lub dwóch warstwach ukrytych i jednej wyjściowej. Uczenie takiego modelu polega na wykonaniu operacji propagacji w przód (model przetwarza dane treningowe i wylicza wartość funkcji błędu) oraz propagacji wstecz (obliczenia wartości gradientu i późniejsze zaktualizowanie wag każdego neurona w oparciu o obliczony gradient i wartość współczynnika uczenia).

W zależności od problemu często zachodzi potrzeba modyfikacji hiperparametrów (np. wartości współczynnika uczenia się) lub funkcji kosztu (np. w przypadku problemów regresyjnych najczęściej stosuje się błąd średniokwadratowy)

Sieci klasy MLP wymagają, by dane miały reprezentację wektorową. W przypadku gdy dane nie mają takiej struktury należy je odpowiednio przekształcić czyli dokonać tzw. ekstrakcji cech. W przypadku języka naturalnego ekstrakcja cech często będzie polegała na tzw. tokenizacji czyli rozbiciu tekstu na sekwencję unikatowo oznaczonych „tokenów", które mogą być potem przetworzone przez sieć neuronową.

Przykładowe zastosowanie

Sieci neuronowe mogą być stosowane do rozwiązywania różnego rodzaju problemów (w tym m. in. problemów regresyjnych i klasyfikacyjnych). Dzięki swojej strukturze są w stanie często uchwycić zaawansowane zależności pomiędzy danymi wejściowymi a wyjściowymi co pozwala im na rozwiązywanie złożonych problemów.

Implementacja

Podczas wykonywania zadania korzystaliśmy z Jupyter Notebook. Po kolei wykonywaliśmy kolejne zadania, które były na liście. Podczas implementacji naszym głównym celem było napisanie takiego kodu, który pozwalałby na łatwe testowanie kolejnych modeli.

Materiały dodatkowe

Do wykonania zadania została użyta instrukcja do zadania, dokumentacja scikit-learn - https://scikit-learn.org/stable/index.html, kod dołączony do zadania oraz opis zadania.

Do wykonania dodatkowego zadania użyliśmy następującego tutoriala - https://scikit-learn.org/stable/auto_examples/neural_networks/plot_mlp_alpha.html

Opis wykorzystanych bibliotek

Do zadania zostały wykorzystane następujące wbudowane biblioteki:

a) os – biblioteka służąca do odnajdywania plików znajdujących się w danym folderze
b) re – biblioteka pozwalająca na stosowanie regex

Oraz następujące dodatkowe biblioteki:

a) BeautifulSoup – biblioteka pozwalająca na wczytywanie html stron internetowych
b) numpy – biblioteka pozwalająca na przetwarzanie wektorów danych
c) pandas – biblioteka pozwalająca na wczytanie danych zapisanych w plikach .csv i ich przetwarzanie
d) matplotlib – biblioteka pozwalająca na tworzenie wykresów
e) sklearn – biblioteka pozwalająca na stosowanie modeli i funkcji związanych z uczeniem maszynowym

f) sentence_transformers – biblioteka pozwalająca na zakodowanie tekstów w postać tzw. embeddingów

Preprocessing danych

Dane otrzymaliśmy w następującej postaci:

a) Folder z plikami html zawierającymi żarty
b) 3 pliki .csv zawierający oceny żartów dokonanych przez użytkowników

Żeby dostać zbiór danych wejściowych podjęliśmy następujące kroki:

a) Wczytaliśmy każdy plik html
b) Z każdego pliku html wyciągnęliśmy żart (było to możliwe, bo przed i po każdym żarcie był odpowiedni komentarz), który zapisaliśmy jako string
c) Z powyższego stringa pozbyliśmy się oznaczeń html (np. <b>)
d) Mając tekst bez elementów html użyliśmy klasy SentenceTransformer z parametrem 'bert-base-cased', żeby przekonwertować tekst na tzw. embedding
e) Wszystkie embeddingi zapisaliśmy w jednej liście otrzymując zbiór danych wejściowych o rozmiarze (100, 768)

Żeby dostać zbiór danych wyjściowych podjęliśmy następujące kroki:

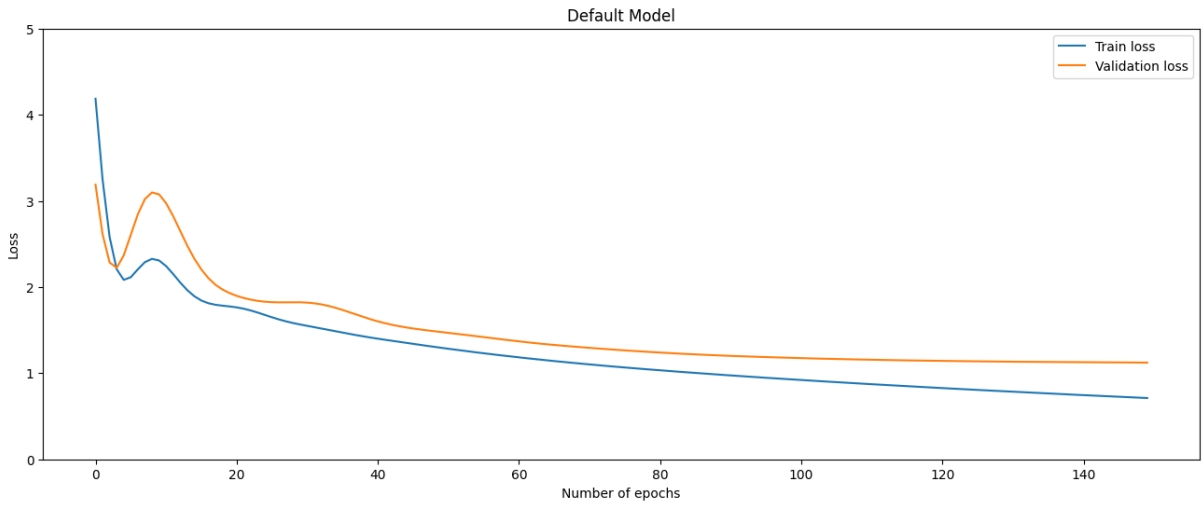
a) Wczytaliśmy każdy plik .csv i je złączyliśmy
b) Zamieniliśmy wszystkie wartości 99.00 na np.nan
c) Dla każdego żartu obliczyliśmy średnią ocen otrzymując zbiór danych wyjściowych o rozmiarze (100)

Warto zauważyć, że maksymalna średnia ocen wynosiła 3.36, a minimalna wynosiła -3.70. Będzie trzeba to wziąć pod uwagę gdy będziemy oceniać poziom śmieszności naszego żartu.

Mając przygotowany zbiór danych wejściowych i wyjściowych przygotowaliśmy zbiór uczący i walidacyjny z użyciem funkcji train_test_split z biblioteki sklearn.

Jak działa podstawowy model MLP?

Przetestowaliśmy nasz model na domyślnym zestawie hiperparametrów. Poniżej znajduje się wykres wartości funkcji kosztu w zależności od numeru epochu z podziałem na koszt treningowy i walidacyjny.



Analizując powyższy wykres jesteśmy w stanie zauważyć następujące rzeczy:

a) Na początku model miał problemy z nauką. Widać to po skoku kosztu treningowego i walidacyjnego w początkowych epochach. Jednak potem już nie było skoków i uczenie przebiegało normalnie.
b) Model dobrze generalizuje się. Widać to po tym, że wraz z kosztem treningowym maleje też koszt walidacyjny.
c) Przy ok. 140 epochu koszt walidacyjny zamiast się zmniejszać zaczyna powoli rosnąć. Oznacza to, że przy tym epochu osiągnęliśmy maksymalną efektywność modelu. Dalsze uczenie modelu doprowadzi do nadmiernego dopasowania.

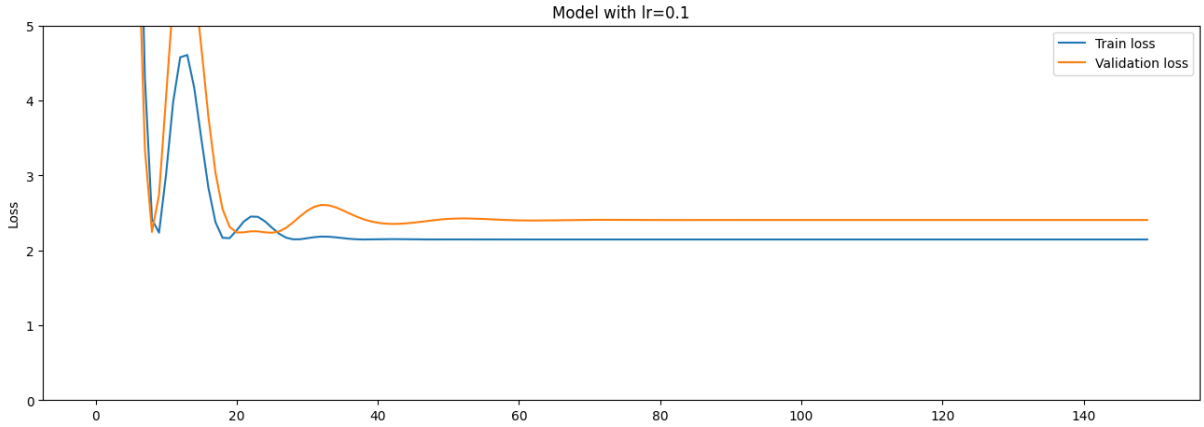
Jak tempo uczenia wpływa na osiągane wyniki?

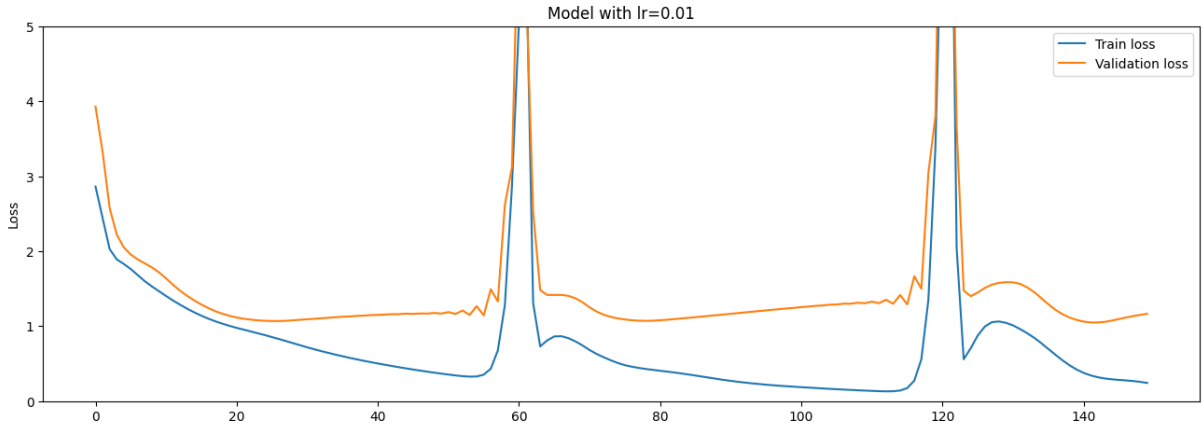
Przetestowaliśmy działanie MLP dla różnych wartości tempa uczenia. Każdy eksperyment został przeprowadzony dla 150 epochów. Wybraliśmy 100 jako liczbę neuronów w warstwie ukrytej.

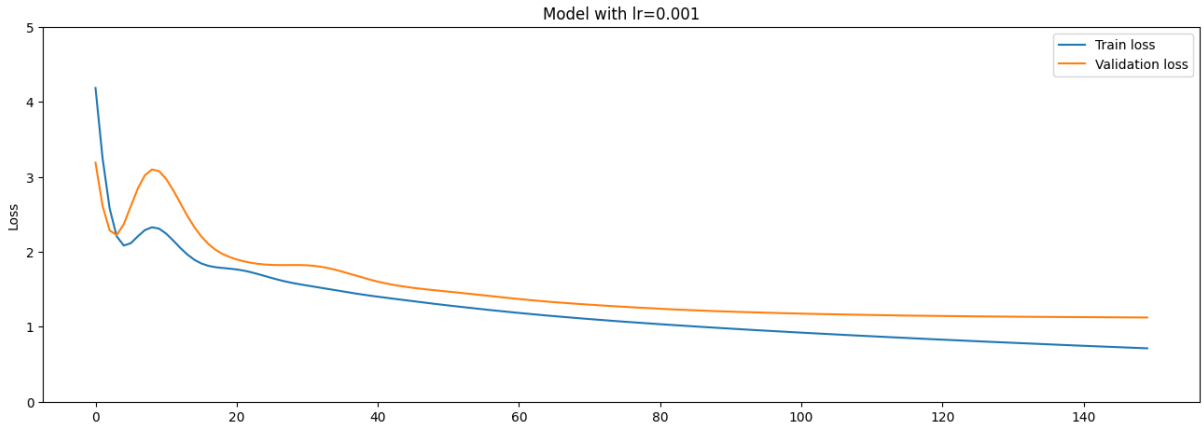
Wybraliśmy następujące wartości tempa uczenia się do eksperymentów:

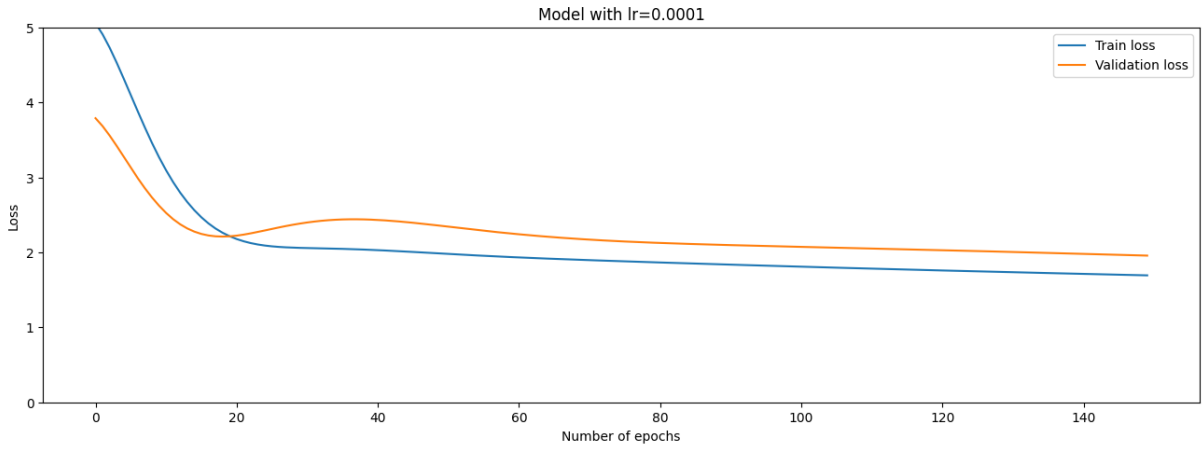
a) 0.1
b) 0.01
c) 0.001 (domyślna wartość hiperparametru)
d) 0.0001

Poniżej znajdują się wyniki eksperymentów


Model with lr=0.1
Train loss
Validation loss
Loss
0
1
2
3
4
5
0
20
40
60
80
100
120
140


Model with lr=0.01
Train loss
Validation loss
Loss
0
1
2
3
4
5
0
20
40
60
80
100
120
140


Model with lr=0.001
Train loss
Validation loss
Loss
0
1
2
3
4
5
0
20
40
60
80
100
120
140


Model with lr=0.0001
Train loss
Validation loss
Loss
0
1
2
3
4
5
0
20
40
60
80
100
120
140
Number of epochs

Patrząc na wyniki eksperymentów możemy wyciągnąć następujące wnioski:

a) Przy zbyt dużym tempie uczenia się proces nauki jest niestabilny. Widzimy to gdy ustawimy wartość tempa uczenia się na 0.1 i 0.01. Na wykresach widzimy nagłe skoki w koszcie treningowym i walidacyjnym. Dodatkowo model ma problem z dopasowaniem się do danych.
b) Przy zbyt małym tempie uczenia się model zbyt wolno się uczy. Możemy to zauważyć gdy ustawimy wartość tempa uczenia się na 0.0001. Widzimy, że model się uczy, ale ma problemy z dopasowaniem się do danych (koszt treningowy i walidacyjny maleje, ale jest on większy od kosztu treningowego i walidacyjnego dla modelu z tempem uczenia ustawionym na 0.001)
c) Wybór odpowiedniej wartości tempa uczenia się jest kluczowy podczas trenowania sieci neuronowych. W przypadku wyboru zbyt dużej wartości uczenie jest niestabilne – mogą występować nagłe wzrosty kosztu treningowego i walidacyjnego. W przypadku wyboru zbyt małej wartości model zbyt wolno się uczy przez co ma problem z dopasowaniem się do danych

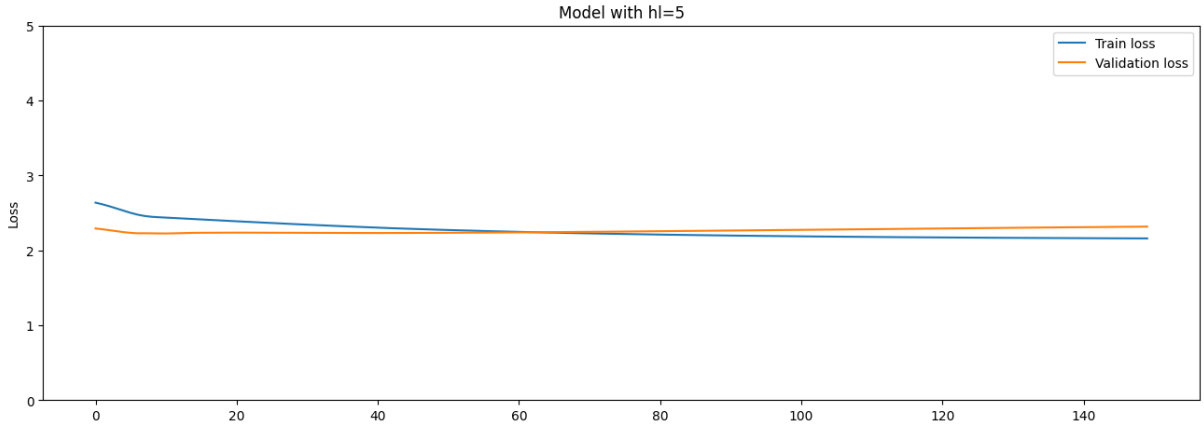
Jak rozmiar modelu MLP wpływa na osiągane wyniki?

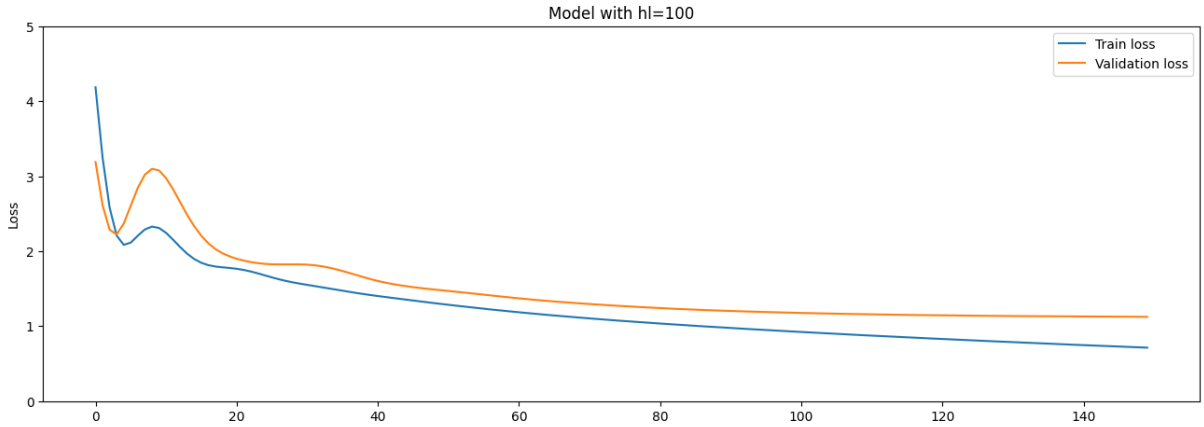
Przetestowaliśmy działanie MLP dla różnej liczbie neuronów w ukrytej warstwie. Każdy eksperyment został przeprowadzony dla 150 epochów. Jako tempo uczenia się wybraliśmy 0.001

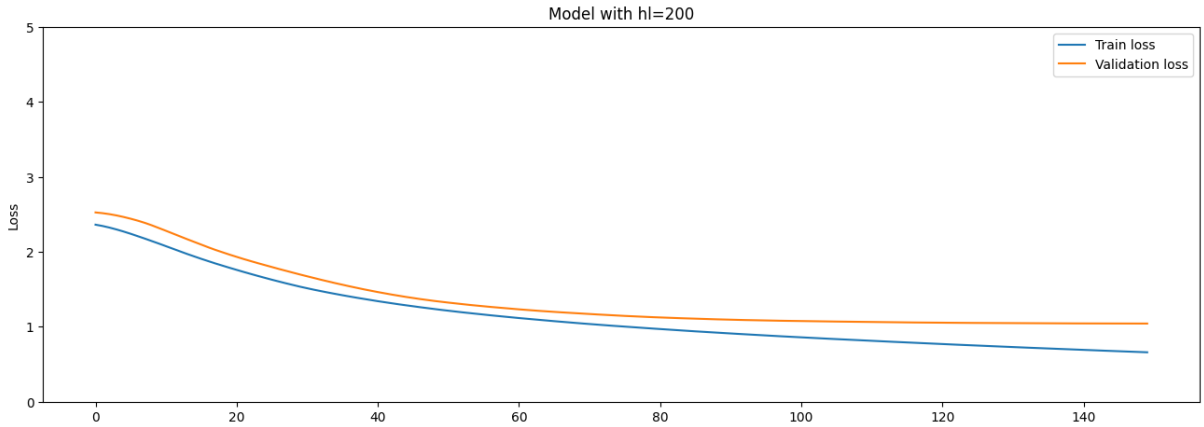
Wybraliśmy następujące liczby neuronów w ukrytej warstwie do eksperymentów:

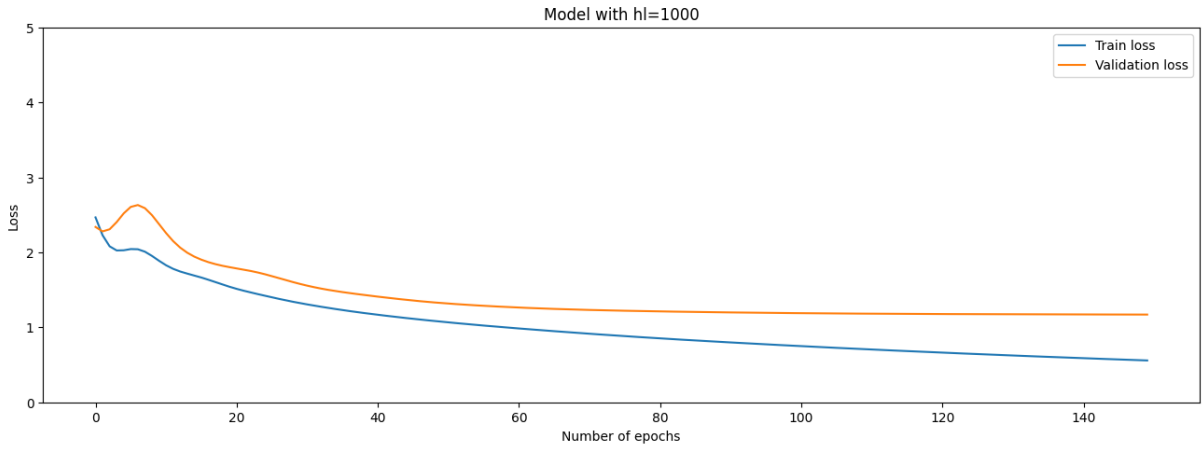
a) 5
b) 100 (domyślna wartość hiperparametru)
c) 200
d) 1000

Poniżej znajdują się wyniki eksperymentów


Model with hl=5
Train loss
Validation loss
Loss
0
1
2
3
4
5
0
20
40
60
80
100
120
140


Model with hl=100
Train loss
Validation loss
Loss
0
1
2
3
4
5
0
20
40
60
80
100
120
140


Model with hl=200
Train loss
Validation loss
Loss
0
1
2
3
4
5
0
20
40
60
80
100
120
140


Model with hl=1000
Train loss
Validation loss
Loss
0
1
2
3
4
5
0
20
40
60
80
100
120
140
Number of epochs

Patrząc na wyniki eksperymentów możemy wyciągnąć następujące wnioski:

a) Przy zbyt dużym rozmiarze modelu, model szybko się uczy, ale często zachodzi zjawisko nadmiernego dopasowania. Widzimy to gdy ustawimy rozmiar modelu na 1000. Od ok. 80 epochu koszt walidacyjny nie maleje, a wręcz zaczyna rosnąć. Koszt treningowy natomiast dalej maleje. Jest duża szansa, że model zamiast uczyć się zależności pomiędzy danymi wejściowymi a wyjściowymi zaczął uczyć się struktury danych treningowych.
b) Przy zbyt małym rozmiarze modelu, model zaczyna mieć problemy dopasowaniem do danych. Widzimy to gdy ustawimy rozmiar modelu na 5. Model nie był w stanie nauczyć się zależności pomiędzy danymi wejściowymi a wyjściowymi co widzimy poprzez wysoki koszt treningowy i walidacyjny
c) Podczas wybierania modelu warto przemyśleć jego rozmiar. Model musi być wystarczająco duży, by móc nauczyć się zależności pomiędzy danymi wejściowymi a wyjściowymi, ale wystarczająco mały, by czasami nie doszło do sytuacji gdzie model uczy się struktury danych treningowych, co prowadzi do nadmiernego dopasowania

Jak regularyzacja wpływa na osiągane wyniki? (zadanie dodatkowe)

Warto zauważyć, że oprócz modyfikacji liczby neuronów w warstwie ukrytej oraz tempa uczenia się możemy też modyfikować inne hiperparametry modelu. Jednym z ciekawszych hiperparametrów jest alpha, które jest odpowiedzialne za regularyzację.

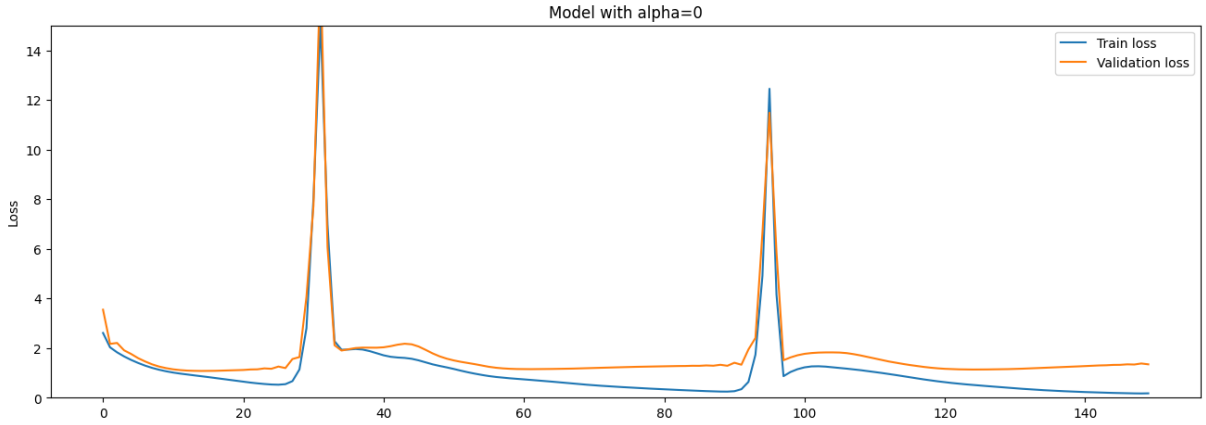
Regularyzacja polega na tym, że do naszego kosztu dodajemy tzw. karę, której celem jest przezwyciężanie nadmiernego dopasowania lub niedopasowania. Zwiększenie wartości parametru alpha naprawia wysoką wariancję modelu (oznakę nadmiernego dopasowania) poprzez zwiększenie znaczenia mniejszych wag. Podobnie zmniejszenie wartości alpha może naprawić wysoki bias (znak niedopasowania do danych) poprzez zwiększenia znaczenia większych wag.

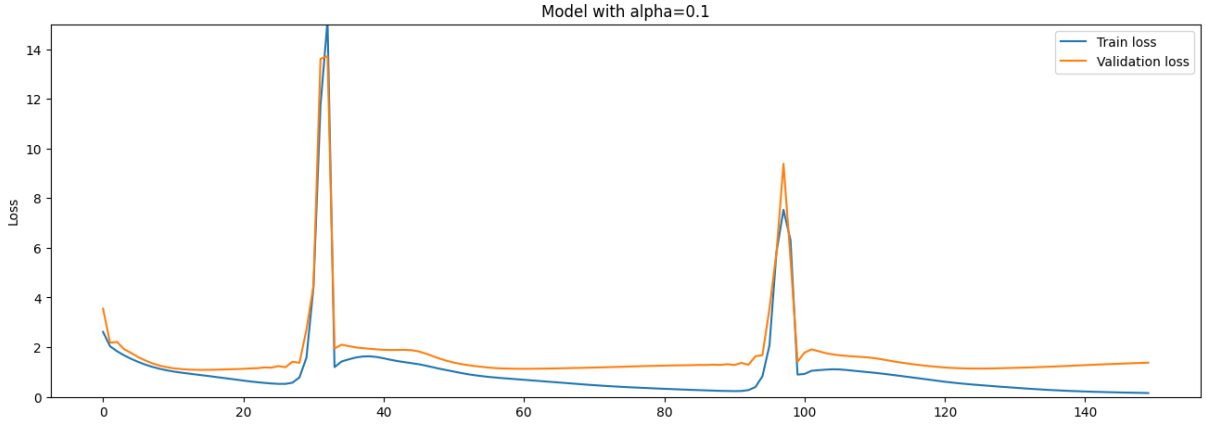
Przetestowaliśmy działanie MLP dla różnej wartości parametru alpha. Każdy eksperyment został przeprowadzony dla 150 epochów. Jako tempo uczenia się wybraliśmy 0.01, a jako liczbę neuronów w ukrytej warstwie wybraliśmy 500.

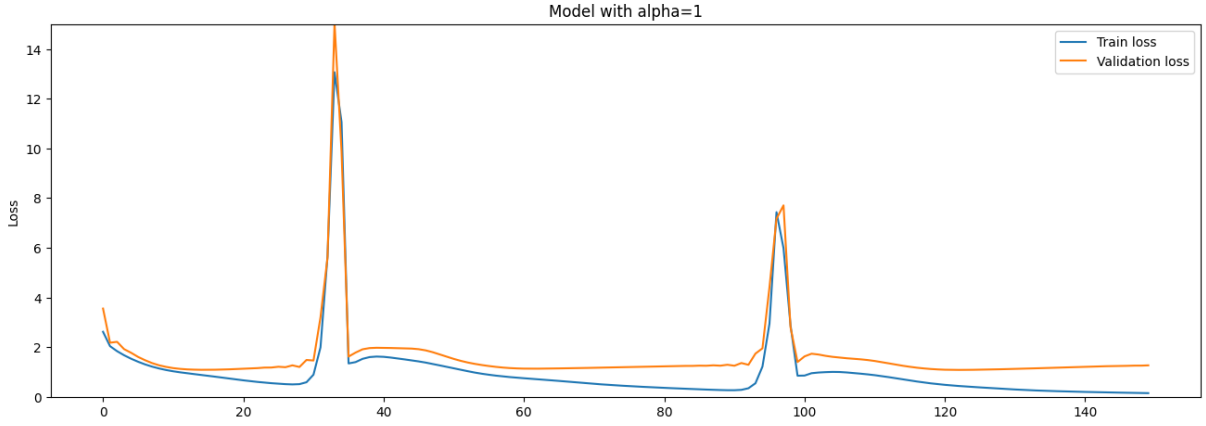
Wybraliśmy następujące wartości parametru alpha do eksperymentów:

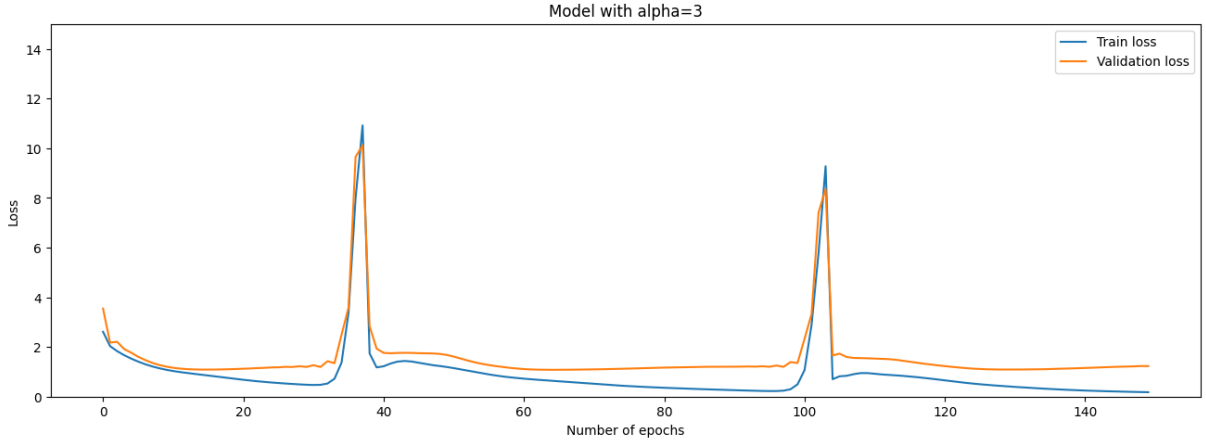
a) 0 (brak regularyzacji)
b) 0.1
c) 1
d) 3

Poniżej znajdują się wyniki eksperymentów


Model with alpha=0
Train loss
Validation loss
Loss


Model with alpha=0.1
Train loss
Validation loss
Loss


Model with alpha=1
Train loss
Validation loss
Loss


Model with alpha=3
Train loss
Validation loss
Loss
Number of epochs

Patrząc na wyniki eksperymentów możemy wyciągnąć następujące wnioski:

a) Tym bardziej zwiększamy alpha tym mniej mamy krzywizn na wykresie kosztu. Wynika to z tego, że zwiększanie wartości alpha zwiększa znaczenie mniejszych wag przez co proces uczenia jest mniej chaotyczny.
b) Warto zauważyć, że modyfikacja wartości alpha nie naprawiła mało efektywnego modelu mimo ustawiania wysokich wartości tego parametru. Jeżeli mamy źle działający model to w pierwszej kolejności powinniśmy modyfikować tempo uczenia się lub liczbę neuronów w warstwie ukrytej. Dopiero gdy te techniki nie dadzą pożądanych rezultatów lub chcemy jeszcze bardziej usprawnić działanie modelu to powinniśmy myśleć o regularyzacji.

Przetestowanie najlepszego modelu

Najlepiej działał w naszych eksperymentach model z tempem uczenia się ustawionym na 0.001 i rozmiarem ustawionym na 200.

Przetestowaliśmy model na następującym tekście „Two aerials get married. The ceremony was rubbish – but the reception was brilliant."

Żeby przekonwertować powyższy tekst na reprezentację wektorową użyliśmy tego samego modelu gdy konwertowaliśmy zbiór danych wejściowych w postać wektorową.

Model przewidział śmieszność naszego żartu na -0.20989. Mając na wglądzie to, że oceny użytkowników były w przedziale [-10, 10] możemy stwierdzić, że żart nie był ani śmieszny ani nieśmieszny. Jest to zgodne z przewidywaniami gdyż moim zdaniem powyższy tekst jest przykładem typowego suchara.

Podczas predykcji warto pamiętać o tym, że poziom śmieszności żartu był bazowany na średniej ocenie użytkowników. Każdy z nas ma inny poziom humoru – coś co dla jednej osoby jest śmieszne dla drugiej może być nieśmieszne. Widać to po tym, że średnie ocen mieszczą się w przedziale [-3,70; 3,36] mimo że oceny mieściły się w przedziale [-10; 10]. Jeżeli by nasz model przewidział ocenę 2.5 dla jakiegoś żartu to oznaczałoby to wówczas, że żart jest śmieszny i niewiele osób uważa, że jest on nieśmieszny.

Podsumowanie

Wykonując zadanie nauczyliśmy się następujących rzeczy:

a) Często w rozwiązywaniu problemów różnego rodzaju używa się sieci neuronowych
b) Przykładem prostej sieci neuronowej jest MLP
c) Trenując model MLP warto monitorować koszt treningowy i walidacyjny, żeby zobaczyć czy model generalizuje się tzn. czy koszt treningowy i walidacyjny maleją wraz z każdą iteracją tzw. epochem
d) Jako hiperparametr możemy ustawić tempo uczenia się. Przy zbyt dużym tempie uczenia się często zachodzi zjawisko niestabilnego uczenia tzn. model ma problem z

dopasowaniem się do danych i często gwałtownie zmienia się koszt treningowy i walidacyjny. Przy zbyt małym tempie uczenia się często zachodzi zjawisko tzw. underfittingu czyli model nie dopasowuje się do danych (ewentualnie dopasowuje się, ale koszt jest znacznie wyższy niż w przypadku identycznego modelu, ale o większym tempie uczenia się)

e) Jako hiperparametr możemy ustawić rozmiar modelu. Przy zbyt dużym modelu często zachodzi zjawisko nadmiernego dopasowania. Przy zbyt małym modelu często zachodzi zjawisko tzw. underfittingu czyli model nie dopasował się do danych.
f) Warto też pamiętać o innych hiperparametrach. Jednym z ciekawszych hiperparametrów jest alpha, które jest odpowiedzialne za regularyzację. Zwiększając wartość alpha naprawiamy wysoką wariancję modelu. Zmniejszając wartość alpha naprawiamy natomiast wysoki bias modelu. Warto jednak pamiętać, że w pierwszej kolejności powinniśmy modyfikować tempo uczenia się i rozmiar modelu, a dopiero potem modyfikować pozostałe hiperparametry. Sama regularyzacja nie naprawi nieefektywnego modelu
g) Podczas wykorzystywania modelu na nowych danych warto pamiętać, że muszą one przejść przez ten sam preprocessing co dane treningowe. Nie można też zapomnieć, że model zwraca tylko wartość – interpretacja co ta wartość znaczy jest już naszym zadaniem