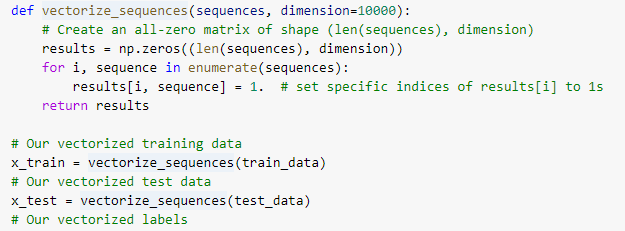
Zadanie 1. Dla Zadania 2 z listy 1 dodaj zastosowanie regularyzacji L1 i L2 oraz porzucania. Przykładowy kod wykorzystania regularyzacji i porzucania w bibliotece Keras: https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/4.4-overfitting-andunderfitting.ipynb. Wypróbuj różne wartości współczynnika regularyzacji i porzucania. Czy wykorzystanie tych mechanizmów pozwoliło zwiększyć skuteczność sieci neuronowej?

Mając dostęp do danych z powyżej wymienionego githuba postanowiłam zagłębić się w treść i nauczyłam się, że można wykonać w inny sposób one-hot-encode.

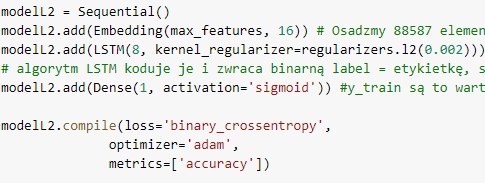
Ja zaimplementowałam one-hot-encode z pomocą zewnętrznej biblioteki: from sklearn.preprocessing import LabelEncoder a ta poniższa to czysty python:



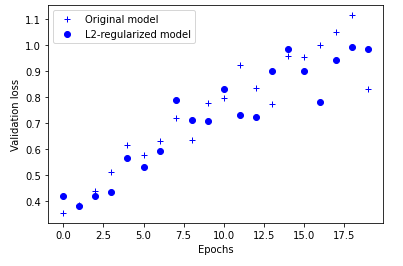
Rys.1 Implementacja definicji funkcji odpowiedzialnej za one-hot-encode

W porównaniu z padingiem, którym posłużyłam się wcześniej, powyższy sposób ma plusy. Dzięki zastosowaniu one-hot-encode na zestawie x, mogę w modelu użyć warstwy Dense już jako pierwsza warstwa. Wcześniej z powodu całkowitych tensorów musiałam użyć warstwy Embedding.

Niespodziewałam się, że w drugiej liście będzie dropout. Ja już go zastosowałam wykonując zadanie 2 z listy pierwszej. Zrobiłam to z ciekawości, dodatkowo. Teraz na potrzeby eksperymentowania z L1 i L2 usunę dropout. W dodatku musiałam zmniejszyć ilość neuronów i zwiększyć liczbę epok, aby zwizualizować wyniki, bo okazało się, że mój wcześniejszy model bardzo szybko osiąga overfitting:



Oto rezultaty:

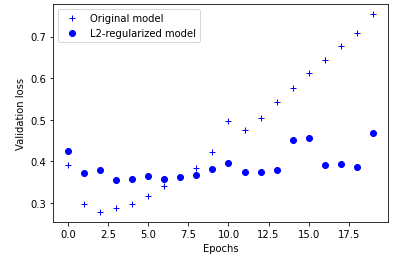


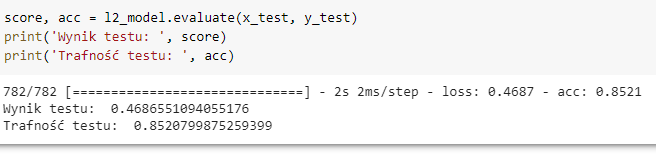
Jak widać w obu przypadkach dochodzi dosyć szybko do overfittingu, dlatego posłużę się nowym modelem z githuba, skoro jest lepszy i na nim przetestuję L1 i jeszcze raz L2 z ich różnymi wartościami.

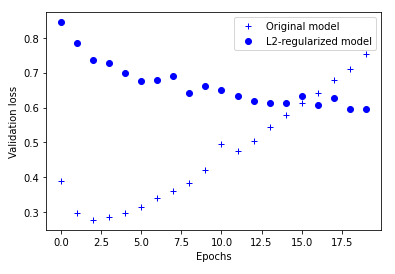
Mimo nienajlepszego modelu to i tak można wydajnie wspomóc się regularyzacja L2.

Model z github, gdzie poszczególne warstwy to Dense jest znacząco szybszy. Jedna epoka trwa 2s, a nie 30s.

Dla l2=0,002 i modelu z githuba wyniki są już znacząco lepsze.



Przyład dla L1=0,001:



Pod względem funkcji straty L1, jak do tej pory, wypada najlepiej. Mimo to zawsze należy pamiętać, że nie tylko to jest istotne. Wystarczy zmniejszyć ilość epok, a już możemy uniknąć overfittingu, a zarazem uzyskać bardzo dobre wyniki funkcji straty, jak i trafności. W powyższym przypadku funkcja straty utrzymuje się ciągle na wysokim poziomie, co nie koniecznie jest pożądanym zjawiskiem. Równocześnie trafność nie przekracza 0,9:

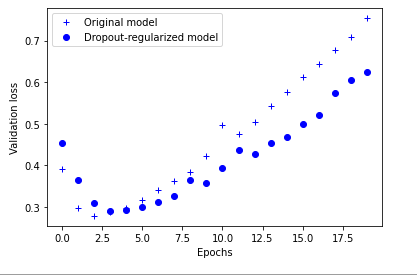


Wnioski:

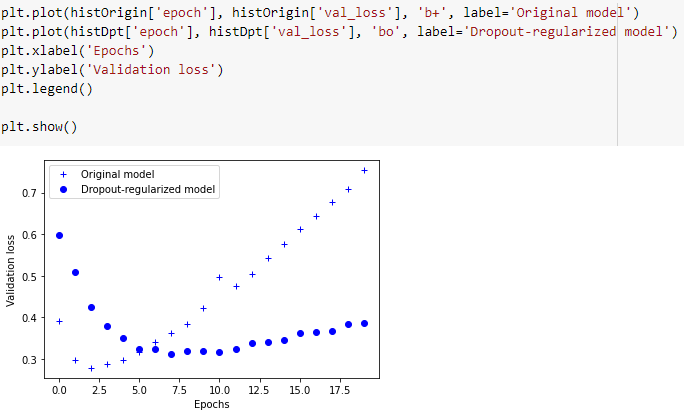
Wykorzystanie wszystkich wymienionych sposobów umożliwiło zminimalizować overfitting i zarazem polepszyło rezultaty sieci neuronowej. Jest to bardzo dobry sposób na zmaksymalizowanie wyników trafności i zminimalizowanie funkcji straty z jednakowym ominięciem przetrenowania.

Zauważyłam też, że tylko w czasie treningu dodajemy karę. W czasie testu wartość loss jest znacznie wyższa.

Dodam od siebie jeszcze dropdown dla sieci neuronowej z github, aby przetestować działanie tego tak jakby dodatkowego szumu, który umożliwia wyeliminować mniej istotne, wręcz konspiracyjne wzorce:



Wyraźna poprawa, a dla wartości większej niż połowa wyjść warstw, tj.:



Zadanie 2. Pobierz z Kaggle zbiór treningowy obrazów https://www.kaggle.com/c/dogs-vs-cats. Korzystając z kodu https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/ 5.2-using-convnets-with-small-datasets.ipynb wybierz 1000 zdjęć psów i 1000 zdjęć kotów do zbioru treningowego, po 500 do zbioru walidacyjnego i po 500 do zbioru testowego [1,2]. Zaproponuj własną strukturę sieci konwolucyjnej, a następnie wytrenuj i sprawdź skuteczność sieci na zbiorze testowym. Jaką skuteczność sieci udało się osiągnąć? Spróbuj poprawić skuteczność sieci korzystając z augmentacji danych. Dla jakich parametrów augmentacji rezultaty są najlepsze?