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:

```
# Our vectorized labels
```

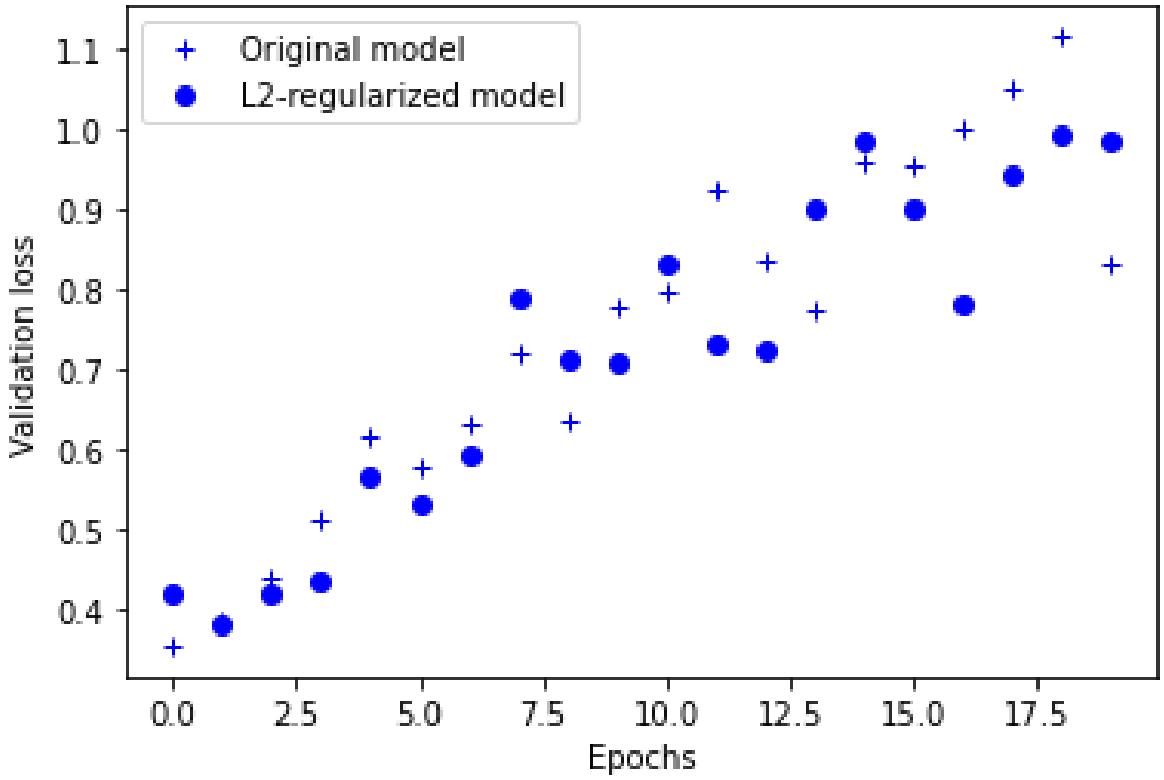
Rys.1 Implementacja definicji funkcji odpowiedzialnej za one-hot-encode

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:

```
modelL2 = Sequential()
modelL2.add(Embedding(max_features, 16)) # Osadzmy 88587 elemen
modelL2.add(LSTM(8, kernel_regularizer=regularizers.l2(0.002)))
# algorytm LSTM koduje je i zwraca binarną label = etykietkę, s
modelL2.add(Dense(1, activation='sigmoid')) #y_train są to wart

modelL2.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

Oto rezultaty:

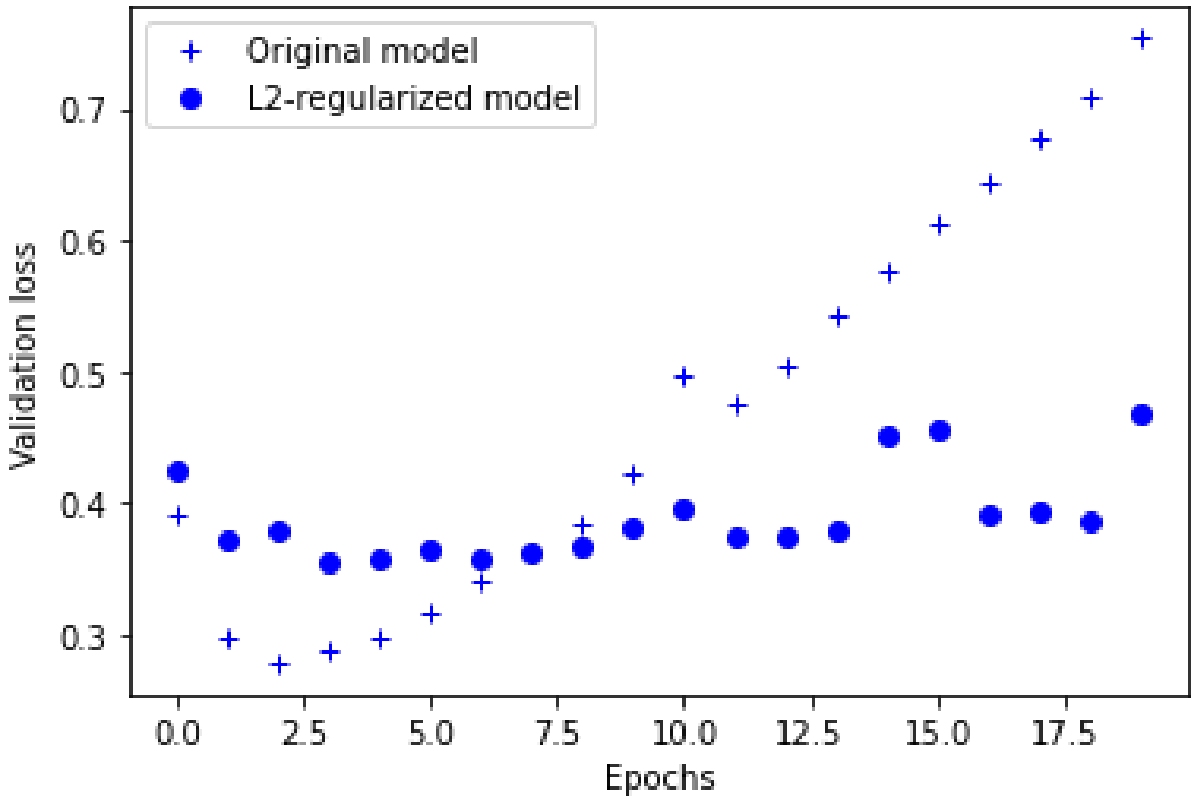


Jak widać w obu przypadkach dochodzi dosyć szybko do overfittingu, dlatego posłużę się nowym modelem z githuba, skoro jest lepszy i na nim przetestuję L1 i jeszcze raz L2 z ich różnymi wartościami.

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.

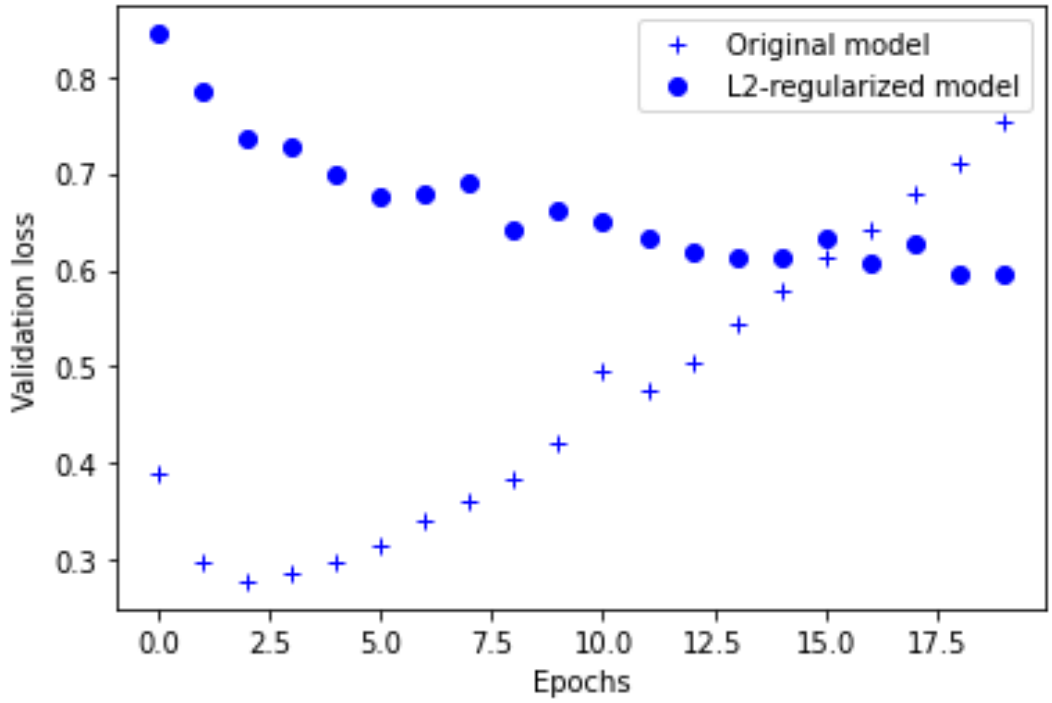
Dla l2=0,002 i modelu z githuba wyniki są już znacząco lepsze.




```
score, acc = l2_model.evaluate(x_test, y_test)
print('Wynik testu: ', score)
print('Trafność testu: ', acc)
```

```
782/782 [==============================] - 2s 2ms/step - loss: 0.4687 - acc: 0.8521
Wynik testu:  0.4686551094055176
Trafność testu:  0.8520799875259399
```

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:

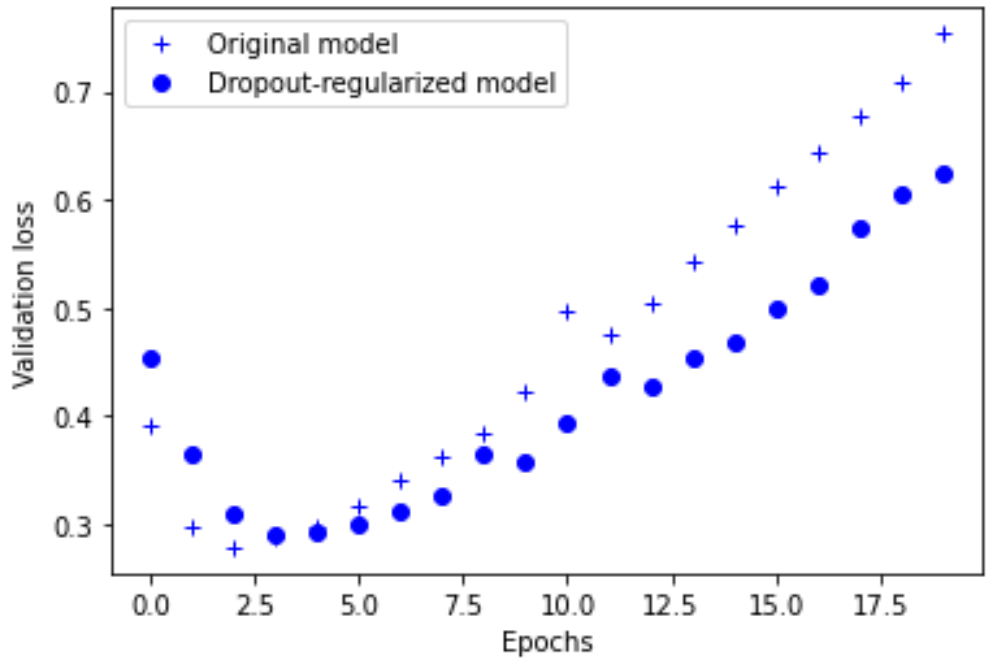
```
782/782 [==============================] - 1s 2ms/step - loss: 0.5969 - acc: 0.8694
Wynik testu:  0.596889853477478
Trafność testu:  0.8694000244140625
```

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.

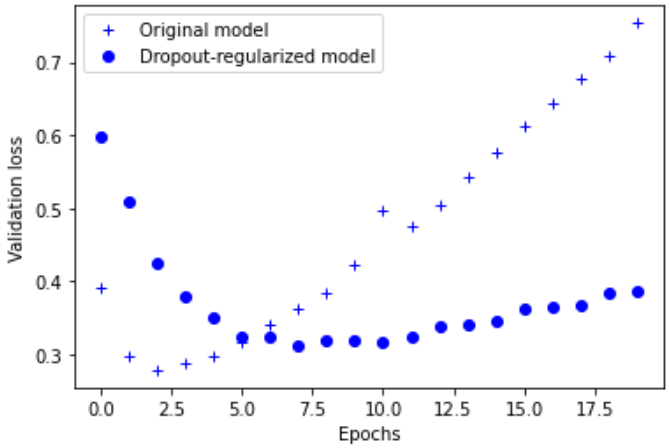
Zauważyłam też, że tylko w czasie treningu dodajemy karę. W czasie testu wartość loss jest znacznie wyższa.

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.:

```
plt.plot(histOrigin['epoch'], histOrigin['val_loss'], 'b+', label='Original model')
plt.plot(histDpt['epoch'], histDpt['val_loss'], 'bo', label='Dropout-regularized model')
plt.xlabel('Epochs')
plt.ylabel('Validation loss')
plt.legend()

plt.show()
```



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?

Poniżej przedstawię serie kodów, które doprowadziły do idealnego podziału, który wymagany jest w treści zadania oraz zadbałam o przetasowanie zbiorów (oczywiście).

```
train_dir = "./train/"
test_dir = "./test1/"

Dtrain = os.listdir(train_dir)
categories = []
for filename in Dtrain:
    category = filename.split(".")[0]
    if category == "dog":
        categories.append("dog")
    else:
        categories.append("cat")
df = pd.DataFrame({
    "filename" : Dtrain,
    "category" : categories
})
```

Rys.1 Implementacja pobierania nazw zdjęć i nadanie im kategorii do struktury danych:DataFrame.

Z DataFrame, pochodzącej z biblioteki pandas, wygodnie mi się pracuje i sądzę, że podziały dokonywane na niej są bardziej przejrzyste, niż na array z biblioteki numpy.

```
        filename category
0  dog.10107.jpg      dog
1   dog.1776.jpg      dog
2  dog.10210.jpg      dog
3  cat.10054.jpg      cat
4  cat.10765.jpg      cat
           filename category
24995  cat.11709.jpg      cat
24996   cat.1247.jpg      cat
24997   cat.8531.jpg      cat
24998   cat.8203.jpg      cat
24999   cat.7937.jpg      cat
```

+ Code + Markdown

```
dfCat = df[df["category"] == 'cat'].head(2000)
dfTestCat = dfCat.head(500)
dfCat = dfCat.iloc[500:,]
dfValCat = dfCat.head(500)
dfCat = dfCat.iloc[500:,]
print(dfCat.shape)
print(dfValCat.shape)
```

```
(1000, 2)
(500, 2)
```

Rys.2 Wymagane podziały dla klasy ‘cat’.

Dla psów zrobiłam identycznie, następnie połączyłam zbiory i przetasowałam je.

```
frames = [dfCat, dfDog]
frames2 = [dfValCat, dfValDog]
frames3 = [dfTestCat, dfTestDog]
```

+ Code + Markdown

```
df2 = pd.concat(frames)
df2Val = pd.concat(frames2)
df2Test= pd.concat(frames3)
print(df2.head())
print(df2.tail())
```

Rys.3 Połączenie danych obu klas razem.

Przedstawiam model, od którego mniej więcej rozpoczęłam kalibracje hiperparametrów, jak i ilość i rodzaj warstw. W komentarzu do kodu określiłam co oznaczają poszczególne wartości.

```
# I need shuffle this data
#The frac keyword argument specifies the fraction of rows to return in the random sample, so frac=1 means return all rows (in random order).
#Here, specifying drop=True prevents .reset_index from creating a column containing the old index entries.
df2 = df2.sample(frac=1).reset_index(drop=True)
df2Val = df2Val.sample(frac=1).reset_index(drop=True)
df2Test = df2Test.sample(frac=1).reset_index(drop=True)
```

Rys.4 Przetasowanie gotowych zbiorów danych.

```
df2
```

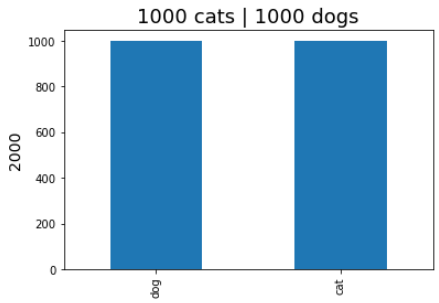
	filename	category
0	cat.8755.jpg	cat
1	cat.5007.jpg	cat
2	cat.10582.jpg	cat
3	cat.6902.jpg	cat
4	cat.7714.jpg	cat
...	...	...
1995	cat.11990.jpg	cat
1996	dog.5491.jpg	dog
1997	dog.8658.jpg	dog
1998	cat.12080.jpg	cat
1999	cat.3969.jpg	cat

2000 rows × 2 columns

Rys.5 Skrócona wizualizacja efektów.

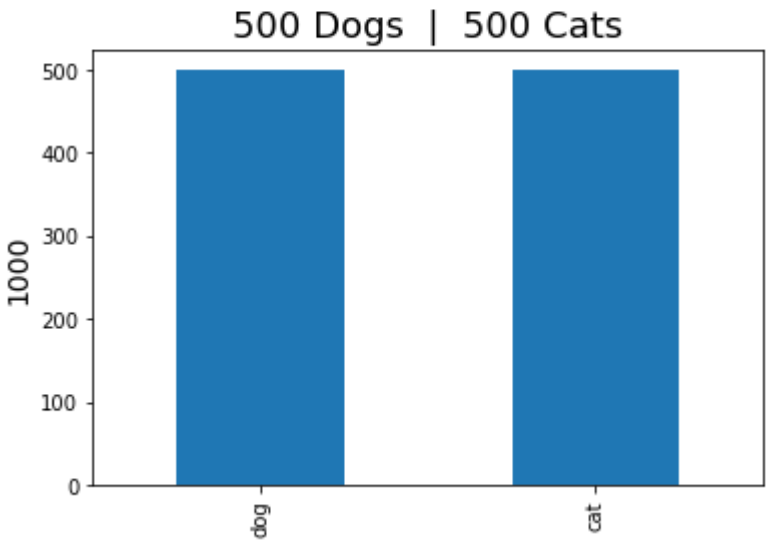
Udowodnienie, że podział jest idealnie równy:

```
df2.category.value_counts().plot.bar()
plt.title(str(len(df2[df2['category'] == 'cat']))+" cats" + " | " +str(len(df2[df2['category'] == 'dog']))+" dogs", fontsize=18)
plt.ylabel(len(df2), fontsize = 14)
plt.show()
```



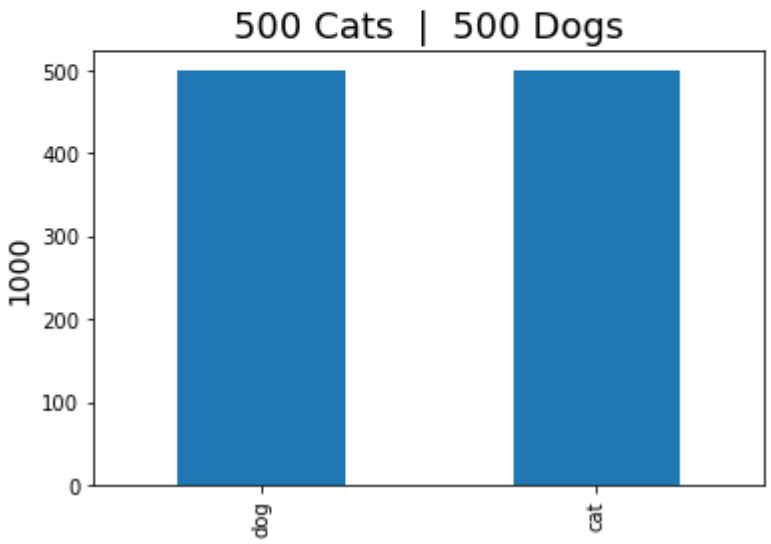
Rys.6 Ilościowy podział danych ze względu na klasę w zestawie uczącym.

```
df2Val.category.value_counts().plot.bar()
plt.ylabel(len(df2Val), fontsize = 14)
plt.title(str(len(df2Val[df2Val['category'] == '
plt.show()
```



Rys.7 Ilościowy podział danych ze względu na klasę w zestawie walidacyjnym.

```
df2Test.category.value_counts().plot.bar()
plt.ylabel(len(df2Test), fontsize = 14)
plt.title(str(len(df2Test[df2Test['category'] ==
plt.show()
```



Rys.8 Ilościowy podział danych ze względu na klasę w zestawie testowym.

```
import pandas as pd
main_dir = "/kaggle/working/"
train_dir = "train"
path = os.path.join(main_dir,train_dir)
convert = lambda category : int(category == 'dog')
X=[]
y=[]


def create_test_data(path):
    for t in df2['filename']:
        print(t)
        category = t.split(".")[0]
        category = convert(category)
        img_array = cv2.imread(os.path.join(path,t),cv2.IMREAD_GRAYSCALE)
        print(img_array)
        new_img_array = cv2.resize(img_array, dsize=(80, 80))
        X.append(new_img_array)
        y.append(category)
```

```
create_test_data(path)
X_train = np.array(X).reshape(-1, 80,80,1)
y_train = np.array(y)
```

```
cat.8755.jpg
[[ 64  63  63 ...  65  67  69]
 [ 67  67  66 ...  56  59  61]
 [ 66  66  65 ...  68  72  76]
```

Rys.9 Kod implementujący zaciąganie danych od zmiennej będącej nazwą obrazu i po adresie katalogu.

```
#Normalize data - as I c
X_train = X_train/255.0
X_test = X_test/255.0
X_val = X_val/255.0
```

+ Code + Markdown

```
print(X_train[1])
```

```
[[[0.05882353]
  [0.05098039]
  [0.08235294]
```

Rys.10 Normalizacja danych wejściowych.

Jest to obraz, każdy piksel zakodowany jest od 0 do 255. Są to duże dane dla komutera. O wiele szybciej i wydajniej pracuje on na mniej rozbieżnych danych jak od 0 do 1, będących zmiennoprzecinkowe.

```
import keras
from keras.utils import np_utils

num_classes=len(np.unique(y_train))
y_train=keras.utils.to_categorical(y_train, num_classes)
y_test=keras.utils.to_categorical(y_test,num_classes)
y_val=keras.utils.to_categorical(y_val,num_classes)
```

Rys.11 Kod implementujący one-hot kodowanie dla zestawów wyjściowych.

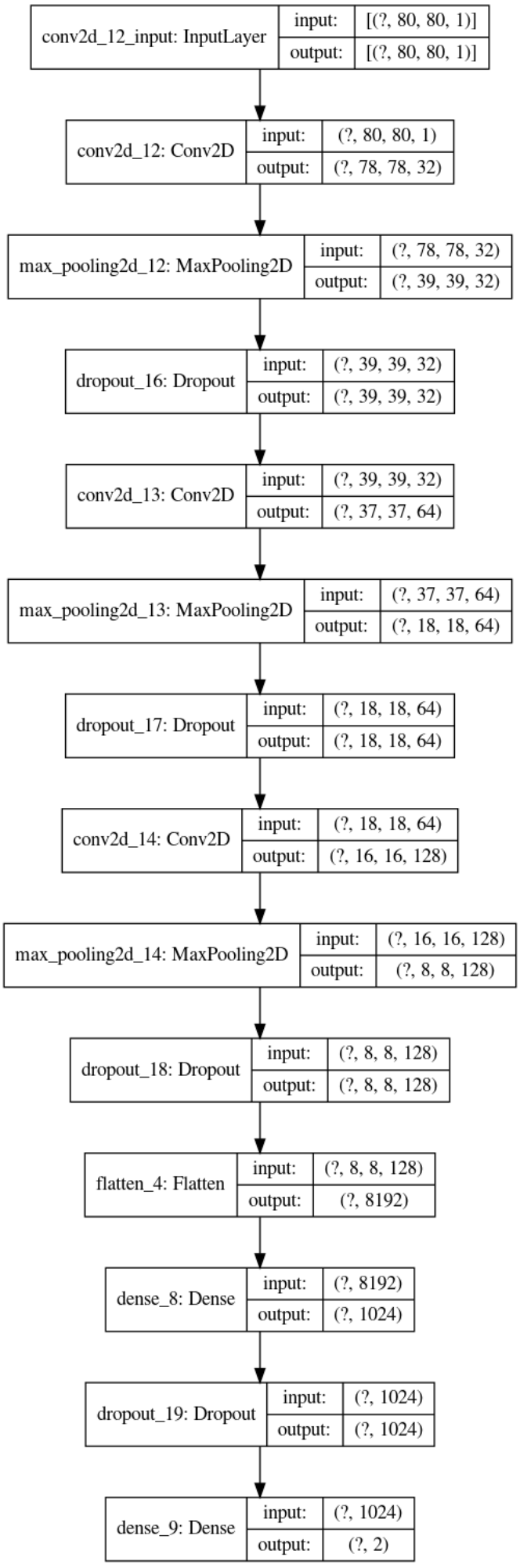
Wykorzystywałam go tylko dla loss=’categorical-crossentropy’.

```
model = Sequential()
model.add(Conv2D(32,(3,3), activation = 'relu', input_shape = X_train.shape[1:]))
model.add((MaxPooling2D(2,2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3,3), activation="relu"))
model.add((MaxPooling2D(2,2)))
model.add(Dropout(0.25))

model.add(Conv2D(128, (3,3), activation="relu"))
model.add((MaxPooling2D(2,2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(1024, activation="relu"))
model.add(Dropout(0.25))

model.add(Dense(2, activation="sigmoid"))

model.compile(optimizer="adam",
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

Rys.12 Implementacja pierwszej wersji modelu.

Powyżej wkradł się błąd. Funkcja loss powinna równać się ‘categorical=crossentropy’, aby po one-hot-encodingu nie wystąpił błąd.

Wiadomo, w przypadku klasyfikacji binarnej, gdzie wybieramy między bitem 0, a 1 nadajemy loss = ‘binary-crossentropy’. Po one-hot-encode krotki tablicy y_train są tylowymiarowe ile jest klas.


conv2d_12_input: InputLayer
input: [(?, 80, 80, 1)]
output: [(?, 80, 80, 1)]
conv2d_12: Conv2D
input: (?, 80, 80, 1)
output: (?, 78, 78, 32)
max_pooling2d_12: MaxPooling2D
input: (?, 78, 78, 32)
output: (?, 39, 39, 32)
dropout_16: Dropout
input: (?, 39, 39, 32)
output: (?, 39, 39, 32)
conv2d_13: Conv2D
input: (?, 39, 39, 32)
output: (?, 37, 37, 64)
max_pooling2d_13: MaxPooling2D
input: (?, 37, 37, 64)
output: (?, 18, 18, 64)
dropout_17: Dropout
input: (?, 18, 18, 64)
output: (?, 18, 18, 64)
conv2d_14: Conv2D
input: (?, 18, 18, 64)
output: (?, 16, 16, 128)
max_pooling2d_14: MaxPooling2D
input: (?, 16, 16, 128)
output: (?, 8, 8, 128)
dropout_18: Dropout
input: (?, 8, 8, 128)
output: (?, 8, 8, 128)
flatten_4: Flatten
input: (?, 8, 8, 128)
output: (?, 8192)
dense_8: Dense
input: (?, 8192)
output: (?, 1024)
dropout_19: Dropout
input: (?, 1024)
output: (?, 1024)
dense_9: Dense
input: (?, 1024)
output: (?, 2)

Rys.13 Wizualizajca modelu pierwszego.

Zaczynam od opisu pierwszego modelu ponieważ ostatni, najlepszy, dla mnie niestety nadal jest dość niesatysfakcjonujący.

Uzyskiwałam takie oto wyniki, pomimo dodania Dropdown:

```
Epoch 1/20
63/63 [==============================] - 17s 264ms/step - loss: 0.7204 - accuracy: 0.4820 - val_loss: 0.6942 - val_accuracy: 0.5000
Epoch 2/20
63/63 [==============================] - 16s 257ms/step - loss: 0.6943 - accuracy: 0.5135 - val_loss: 0.6920 - val_accuracy: 0.5120
Epoch 3/20
63/63 [==============================] - 17s 265ms/step - loss: 0.6919 - accuracy: 0.5205 - val_loss: 0.6875 - val_accuracy: 0.5690
Epoch 4/20
63/63 [==============================] - 16s 261ms/step - loss: 0.6816 - accuracy: 0.5610 - val_loss: 0.6660 - val_accuracy: 0.6170
Epoch 5/20
63/63 [==============================] - 16s 257ms/step - loss: 0.6553 - accuracy: 0.6085 - val_loss: 0.6406 - val_accuracy: 0.6320
Epoch 6/20
63/63 [==============================] - 17s 269ms/step - loss: 0.6429 - accuracy: 0.6260 - val_loss: 0.6606 - val_accuracy: 0.6130
Epoch 7/20
63/63 [==============================] - 16s 262ms/step - loss: 0.6209 - accuracy: 0.6745 - val_loss: 0.6287 - val_accuracy: 0.6560
Epoch 8/20
63/63 [==============================] - 16s 258ms/step - loss: 0.6055 - accuracy: 0.6775 - val_loss: 0.6027 - val_accuracy: 0.6860
Epoch 9/20
63/63 [==============================] - 16s 259ms/step - loss: 0.5869 - accuracy: 0.6895 - val_loss: 0.6001 - val_accuracy: 0.6760
Epoch 10/20
63/63 [==============================] - 17s 265ms/step - loss: 0.5339 - accuracy: 0.7360 - val_loss: 0.5957 - val_accuracy: 0.6900
Epoch 11/20
63/63 [==============================] - 17s 262ms/step - loss: 0.5171 - accuracy: 0.7435 - val_loss: 0.5793 - val_accuracy: 0.7030
Epoch 12/20
63/63 [==============================] - 16s 256ms/step - loss: 0.4845 - accuracy: 0.7670 - val_loss: 0.5867 - val_accuracy: 0.6930
Epoch 13/20
63/63 [==============================] - 17s 272ms/step - loss: 0.4321 - accuracy: 0.7915 - val_loss: 0.5932 - val_accuracy: 0.6960
Epoch 14/20
63/63 [==============================] - 16s 259ms/step - loss: 0.3875 - accuracy: 0.8215 - val_loss: 0.5888 - val_accuracy: 0.6970
Epoch 15/20
63/63 [==============================] - 16s 259ms/step - loss: 0.3376 - accuracy: 0.8490 - val_loss: 0.6158 - val_accuracy: 0.7010
Epoch 16/20
63/63 [==============================] - 16s 255ms/step - loss: 0.2722 - accuracy: 0.8870 - val_loss: 0.6482 - val_accuracy: 0.6940
Epoch 17/20
63/63 [==============================] - 17s 274ms/step - loss: 0.2295 - accuracy: 0.9055 - val_loss: 0.7290 - val_accuracy: 0.7060
Epoch 18/20
63/63 [==============================] - 16s 254ms/step - loss: 0.1772 - accuracy: 0.9320 - val_loss: 0.8433 - val_accuracy: 0.7000
Epoch 19/20
63/63 [==============================] - 16s 257ms/step - loss: 0.1515 - accuracy: 0.9440 - val_loss: 0.8271 - val_accuracy: 0.7100
Epoch 20/20
63/63 [==============================] - 16s 257ms/step - loss: 0.1339 - accuracy: 0.9515 - val_loss: 0.9010 - val_accuracy: 0.7000
```

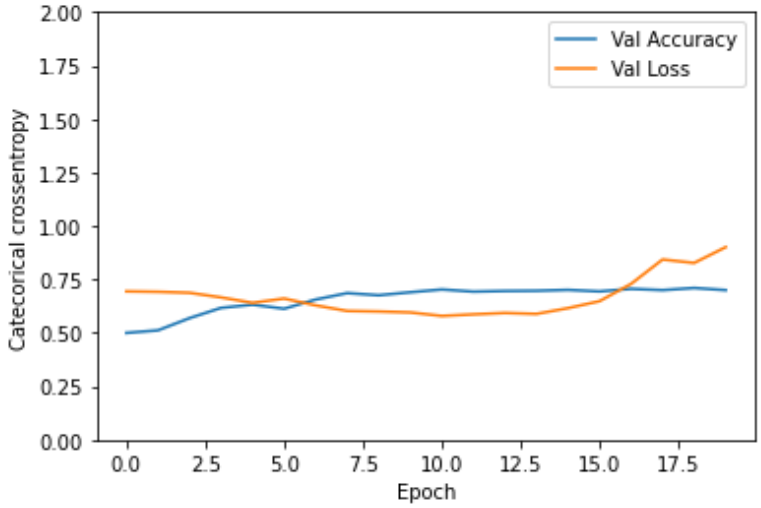
Rys.14 Rezultaty trenowania modelu pierwszego.

Jak widać poniżej, pomimo nawet wysokiej trafności, błąd straty jest bardzo duży. Z czego to wynika? Odpowiem we wnioskach na samym końcu sprawozdania.

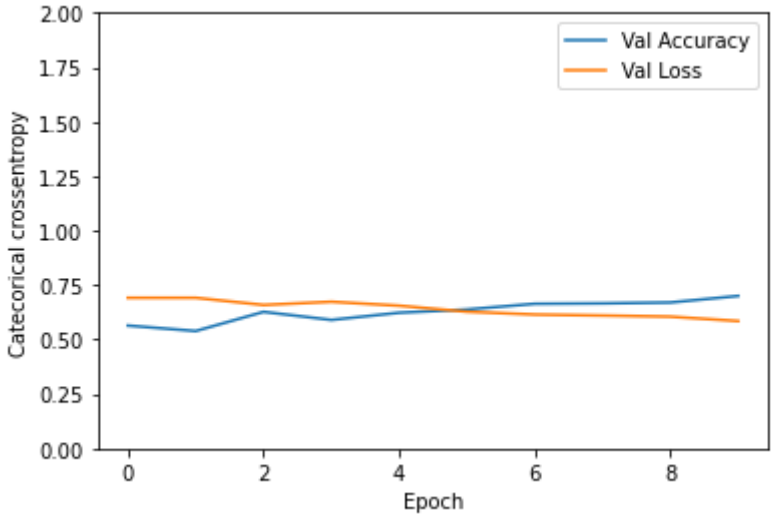
```
32/32 [==============================] - 1s 42ms/step - loss: 0.8978 - accuracy: 0.7040

 Test_Accuracy:- 0.7039999961853027
```

Rys.15 Trafność dla zbioru testowego modelu pierwszego.

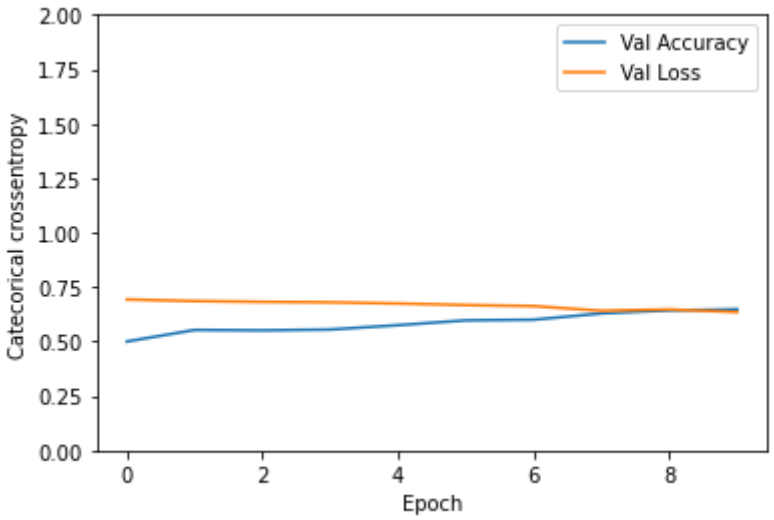
```python
import matplotlib.pyplot as plt

def plot_history():
    plt.figure()
    plt.xlabel('Epoch')
    plt.ylabel('Catecorical crossentropy')
    plt.plot(hist['epoch'], hist['val_accuracy'], label='Val Accuracy')
    plt.plot(hist['epoch'], hist['val_loss'], label = 'Val Loss')
    plt.legend()
    plt.ylim([0,2])

plot_history()
```



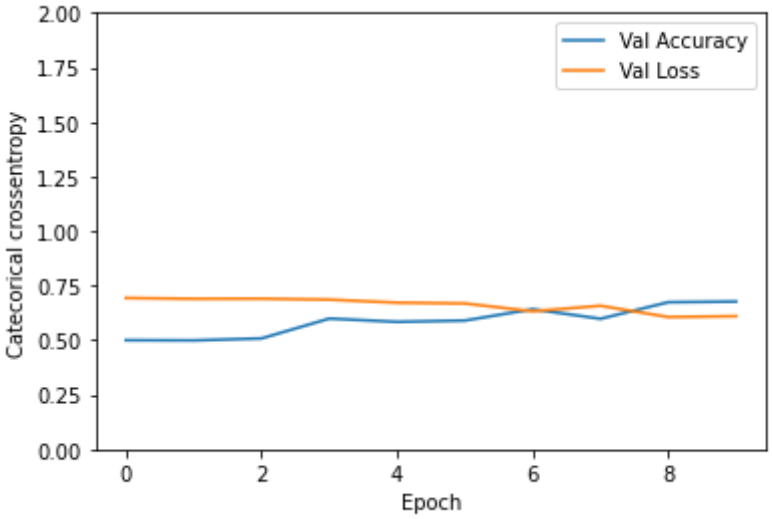
Rys.16 Wizualizacja wykresu funkcji straty i trafności dla zestawu walidacyjnego modelu pierwszego.



Rys.17 Kolejna próba poprawienia rezultatów.



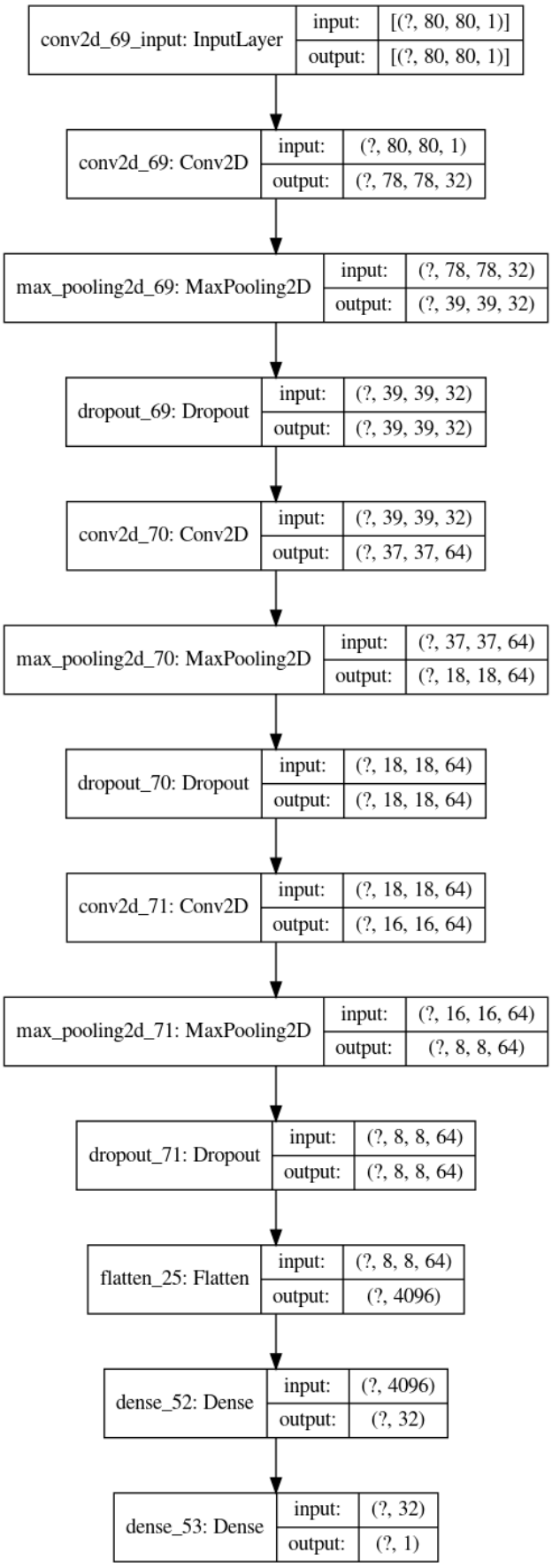
Rys.18 Kolejna próba poprawienia rezultatów.



Rys. 19 Wizualizacja ostatecznych wyników modelu przedstawionego poniżej.

```
model = Sequential()
model.add(Conv2D(32,(3,3), activation = 'relu', input_shape = X_train.shape[1:]))
model.add((MaxPooling2D(2,2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3,3), activation="relu"))
model.add((MaxPooling2D(2,2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3,3), activation="relu"))
model.add((MaxPooling2D(2,2)))
model.add(Dropout(0.5))

model.add(Flatten())
model.add(Dense(32, activation="relu"))

model.add(Dense(1, activation="sigmoid"))

model.compile(optimizer="adam",
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

Rys.20 Implementacja ostatecznego modelu.



Rys.21 Wizualizacja ostatniego modelu.

```
history= model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_val,y_val))
```

```
Epoch 1/10
63/63 [==============================] - 11s 176ms/step - loss: 0.6954 - accuracy: 0.4940 - val_loss: 0.6932 - val_accuracy: 0.5000
Epoch 2/10
63/63 [==============================] - 11s 179ms/step - loss: 0.6931 - accuracy: 0.5205 - val_loss: 0.6896 - val_accuracy: 0.4990
Epoch 3/10
63/63 [==============================] - 11s 181ms/step - loss: 0.6925 - accuracy: 0.4970 - val_loss: 0.6900 - val_accuracy: 0.5080
Epoch 4/10
63/63 [==============================] - 11s 178ms/step - loss: 0.6909 - accuracy: 0.5280 - val_loss: 0.6858 - val_accuracy: 0.5990
Epoch 5/10
63/63 [==============================] - 11s 179ms/step - loss: 0.6812 - accuracy: 0.5580 - val_loss: 0.6718 - val_accuracy: 0.5840
Epoch 6/10
63/63 [==============================] - 11s 179ms/step - loss: 0.6562 - accuracy: 0.6140 - val_loss: 0.6683 - val_accuracy: 0.5900
Epoch 7/10
63/63 [==============================] - 11s 173ms/step - loss: 0.6419 - accuracy: 0.6395 - val_loss: 0.6324 - val_accuracy: 0.6420
Epoch 8/10
63/63 [==============================] - 12s 183ms/step - loss: 0.6188 - accuracy: 0.6560 - val_loss: 0.6577 - val_accuracy: 0.5980
Epoch 9/10
63/63 [==============================] - 12s 195ms/step - loss: 0.6003 - accuracy: 0.6775 - val_loss: 0.6061 - val_accuracy: 0.6740
Epoch 10/10
63/63 [==============================] - 11s 180ms/step - loss: 0.5889 - accuracy: 0.6925 - val_loss: 0.6099 - val_accuracy: 0.6770
```

Rys.22 Rezultaty trenowania ostatniego modelu.

```
accuracy = model.evaluate(X_test, y_test, verbose = 1)
print('\n','Test_Accuracy:-', accuracy[1])
```

```
32/32 [==============================] - 1s 32ms/step - loss: 0.6177 - accuracy: 0.6850

 Test_Accuracy:- 0.6850000023841858
```

Rys.23 Trafność dla zestawu testowego modelu ostatniego.

```
hist = pd.DataFrame(history.history)
print(hist)
hist['epoch'] = history.epoch
```

	loss	accuracy	val_loss	val_accuracy
0	0.695449	0.4940	0.693210	0.500
1	0.693080	0.5205	0.689562	0.499
2	0.692478	0.4970	0.689962	0.508
3	0.690879	0.5280	0.685827	0.599
4	0.681168	0.5580	0.671816	0.584
5	0.656236	0.6140	0.668288	0.590
6	0.641882	0.6395	0.632443	0.642
7	0.618825	0.6560	0.657665	0.598
8	0.600295	0.6775	0.606070	0.674
9	0.588850	0.6925	0.609934	0.677

Rys.24 Dokładniejsza wizualizacja do grafu funkcji strat dla zestawu walidacyjnego.

Niestety we wszystkich przypadkach dochodziło do overfittingu i niestety nie mogłam temu zapobiec. W ostatnim modelu nie jest najgorzej, ponieważ osiągnęłam trafność na

wysokości prawie 70%, a funkcja straty równa była 61,5% - dla zestawu testowego. Jest to jakiś pozytywny wynik.

Hiperparametry dobierałam na podstawie prób i błędów. Kierowałam się wynikami funkcji straty. Często dochodziło do stabilizacji obu z nich. Zdażało się tak tylko dla loss='categorical-crossentropy', dla binarnej krosentropii wyniki były nieznacznie lepsze.

Następnie dodałam Dropout i zmieniałam ilość warstw i neuronów, na mniejszą, aby nie doszło do przetrenowania. Wtedy trafność też spadała. Dlatego podkalibrowałam ilość wartw z filtrami na zdjęcia i maxpoolout. Myślę, że to poprawiło wyłapywać sieci znaczące różnice między zjęciami kotów i psów. Odkrywało wzorce, ale nie nadinterpretowało je później w warstwie Dense, bo jej ilość zmniejszyłam. Mam nadzieję, że poprawnie myślę.

Poszukałam też odpowiedzi na pytanie: dlacego funkcja loss wzrasta?

Znalazłam kilka możliwych wytłumaczeń, żadne niestety nic nie gwarantuje. Podobno tym m.in. charakteryzuje się uczenie maszynowe. Doświadczeniem i cierpliwością w kalibracji sieci głębokich.

1. Strata i trafność nie są dokładnie odwrotnie skorelowane, ponieważ strata mierzy różnice między surowym przewidywaniem (float), a klasą (int 0 lub 1). Dokładnośćnatomiast mierzy różnice między progowym przewidywaniem (0 lub 1), a klasą. Jeśli więc zmieniają się surowe prognozy, zmienia się strata, ale dokładność jest bardziej „odporna", ponieważ prognozy muszą przekroczyć (over/under) próg, aby faktycznie zmienić dokładność.
2. Rozważając przypadek klasyfikacji binarnej, w której zadaniem jest przewidzenie, czy obraz jest kotem czy psem, a wyjściem sieci jest sigmoida (wyprowadzanie liczby zmiennoprzecinkowej między 0 a 1), gdzie trenujemy sieć 1, jeśli zdjęcie przedstawia kota i 0 w innym przypadku. Uważam, że w tym przypadku dwa zjawiska zachodzą w tym samym czasie.

   Niektóre obrazy z prognozy granicznej są lepiej przewidywane, więc ich klasa wyjściowa zmienia się (np. Obraz kota, którego przewidywanie było równe 0,4, staje się 0,6). Jest to klasyczne zachowanie polegające na zmniejszeniu strat, a zwiększenie dokładności.

   Niektóre obrazy z bardzo złymi przewidywaniami są coraz gorsze (np. Obraz kota, którego przewidywania wynosiły 0,2, zmienia się w 0,1). Prowadzi to do mniej klasycznego „wzrostu strat przy niezmienionej dokładności". Zauważ, że gdy do klasyfikacji wykorzystuje się utratę krzyżowej entropii, jak to zwykle się dzieje, złe prognozy są znacznie bardziej karane niż dobre prognozy. W przypadku obrazu kota strata wynosi log (1-prediction), więc nawet jeśli wiele obrazów kota jest poprawnie przewidzianych (niska strata), pojedynczy błędnie sklasyfikowany obraz kota będzie miał wysoką stratę, **stąd „wysadzenie"** Twojej średniej straty.

3. Rosnąca strata i stabilna dokładność mogą być również spowodowane nieco gorszą klasyfikacją dobrych prognoz, ale wydaje mi się, że jest to mniej prawdopodobne z powodu tej „asymetrii" straty.

4. Myślę więc, że gdy wzrasta zarówno dokładność, jak i straty, sieć zaczyna się nadużywać i oba zjawiska zachodzą w tym samym czasie. Sieć zaczyna uczyć się wzorców istotnych tylko dla zbioru uczącego i nie nadaje się do uogólniania, co prowadzi do zjawiska 2, niektóre obrazy ze zbioru walidacyjnego są przewidywane naprawdę źle, z efektem wzmacnianym przez „asymetrię strat". Jednocześnie jednak wciąż uczy się pewnych wzorców przydatnych do uogólniania (zjawisko pierwsze, „dobre uczenie się"), ponieważ coraz więcej obrazów jest poprawnie klasyfikowanych.
5. Wreszcie, myślę, że efekt ten można jeszcze bardziej zaciemnić w przypadku klasyfikacji wieloklasowej, w której sieć w danej epoce może być znacznie nadwyrężona na niektórych klasach, ale nadal uczyć się na innych. - to dokładnie tłumaczy stabilne wyniki obu funkcji uzyskane dla loss='categorical-crossentropy'.