**Опит 1.**

Използвана архитектура на невронната мрежа- Modified Xception- хибрид между Inception и ResNet от keras.io – source code:

<https://keras.io/examples/vision/image_classification_from_scratch/>

* резолюция на изображенията- 256x256 пиксела, заредени в grayscale
* Batch size – 8 изображения
* Брой изображения за трениране на мрежата – 2668, като е зададен 20% validation split за автоматична валидация след всяка епоха. При всяко ново зареждане на изображенията за трениране в паметта и респективно заделянето на сет за валидация е използван един и същ **seed(1333)-** винаги едни и същи изображения попадат в частта за валидация/трениране
* Използван хардуер за изчисленията – GPU – NVIDIA GeForce 940mx – 2GB ; Memory Bandwidth - 37,33 GiB/s ; ComputeCapability – 5.0

Код на архитектурата на мрежата:

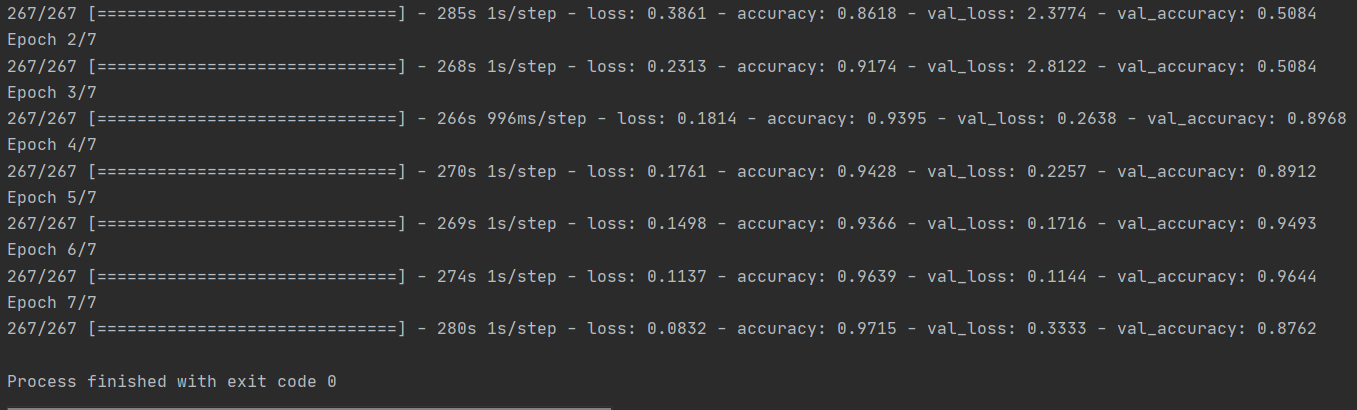
def model\_create\_xception(input\_shape, num\_classes):  
 inputs = krs.Input(shape=input\_shape)  
  
 # Entry block  
 x = krs.layers.experimental.preprocessing.Rescaling(1.0 / 255)(inputs)  
 x = krs.layers.Conv2D(32, 3, strides=2, padding="same")(x)  
 x = krs.layers.BatchNormalization()(x)  
 x = krs.layers.Activation("relu")(x)  
  
 x = krs.layers.Conv2D(64, 3, padding="same")(x)  
 x = krs.layers.BatchNormalization()(x)  
 x = krs.layers.Activation("relu")(x)  
  
 previous\_block\_activation = x # Set aside residual  
  
 for size in [128, 256, 512, 728]:  
 x = krs.layers.Activation("relu")(x)  
 x = krs.layers.SeparableConv2D(size, 3, padding="same")(x)  
 x = krs.layers.BatchNormalization()(x)  
  
 x = krs.layers.Activation("relu")(x)  
 x = krs.layers.SeparableConv2D(size, 3, padding="same")(x)  
 x = krs.layers.BatchNormalization()(x)  
  
 x = krs.layers.MaxPooling2D(3, strides=2, padding="same")(x)  
  
 # Project residual  
 residual = krs.layers.Conv2D(size, 1, strides=2, padding="same")(  
 previous\_block\_activation  
 )  
 x = krs.layers.add([x, residual]) # Add back residual  
 previous\_block\_activation = x # Set aside next residual  
  
 x = krs.layers.SeparableConv2D(1024, 3, padding="same")(x)  
 x = krs.layers.BatchNormalization()(x)  
 x = krs.layers.Activation("relu")(x)  
  
 x = krs.layers.GlobalAveragePooling2D()(x)  
 if num\_classes == 2:  
 activation = "sigmoid"  
 units = 1  
 else:  
 activation = "softmax"  
 units = num\_classes  
  
 x = krs.layers.Dropout(0.5)(x)  
 outputs = krs.layers.Dense(units, activation=activation)(x)  
 return krs.Model(inputs, outputs)

**Run № 1:**

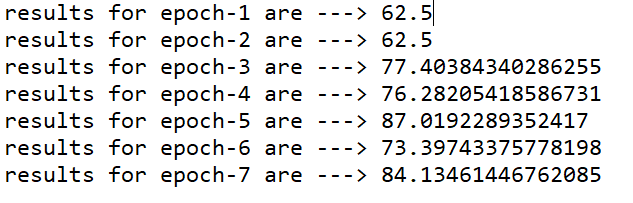
Параметри:

optimizer- Adam(learning\_rate= 0.001) ; loss function- binary crossentropy ; **number of epochs – 7**

Резултати при тренирането:



Резултати при тестването:



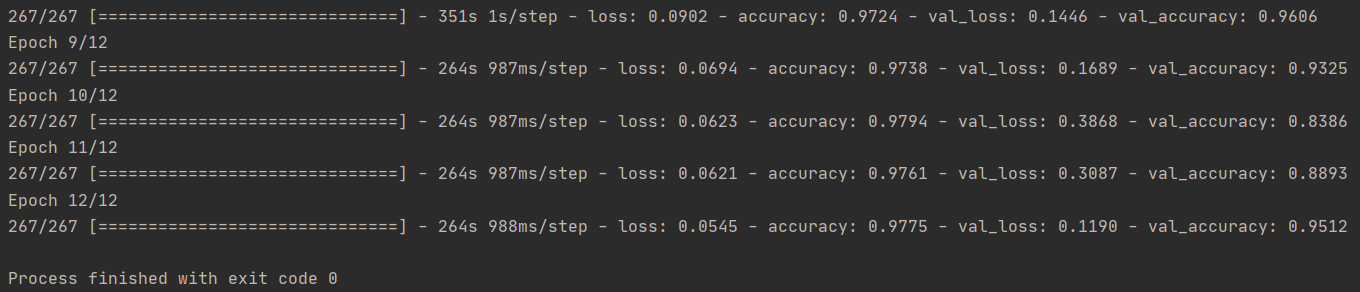
**Run № 2:**

Зареждане на модела от .h5 файл в състояние мрежата- епоха 7 и трениране за още 5 епохи

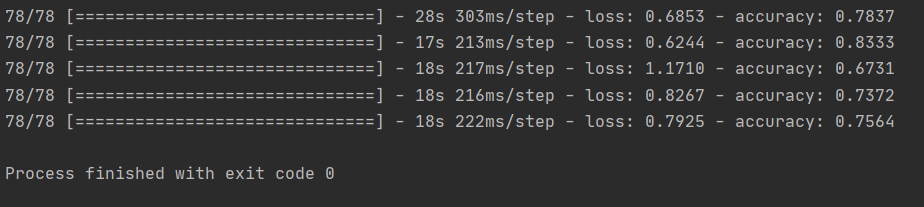
Параметри:

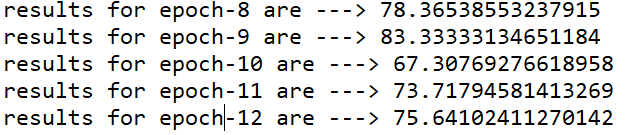
optimizer- Adam(learning\_rate=0.001) ; loss function- binary crossentropy ; **number of epochs – 5(total of 12)**

Резултати при тренирането:

****

Резултати при тестването:



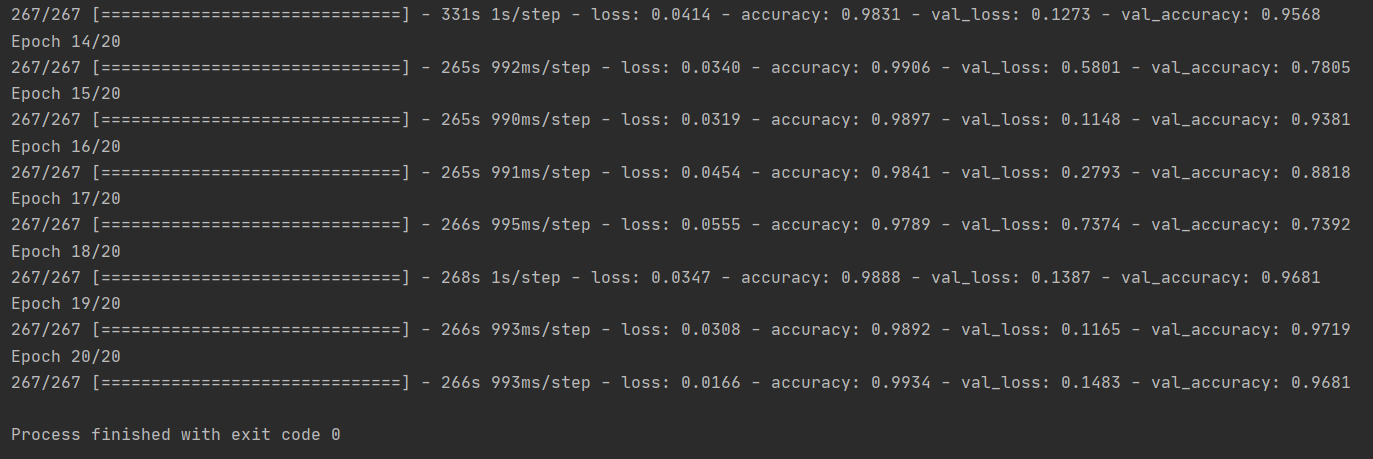


**Run № 3:**

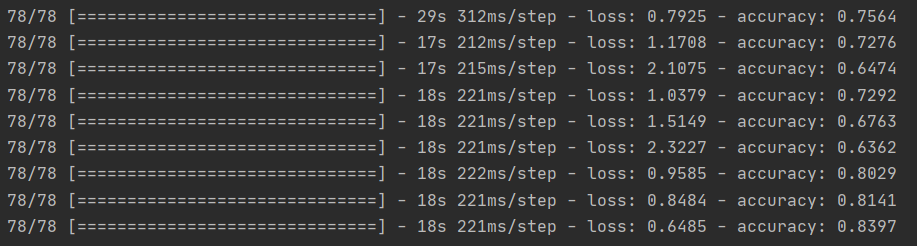
Зареждане на модела от .h5 файл в състояние мрежата- епоха 12 и трениране за още 8 епохи

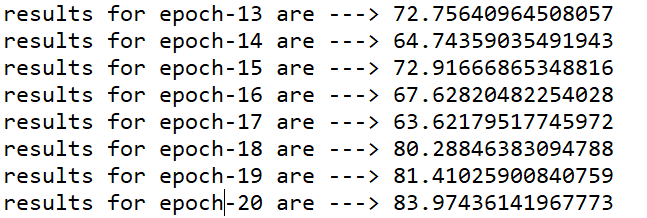
Параметри:

optimizer- Adam(learning\_rate=0.001) ; loss function- binary crossentropy ; **number of epochs – 8(total of 20)**



Резултати от тестването:



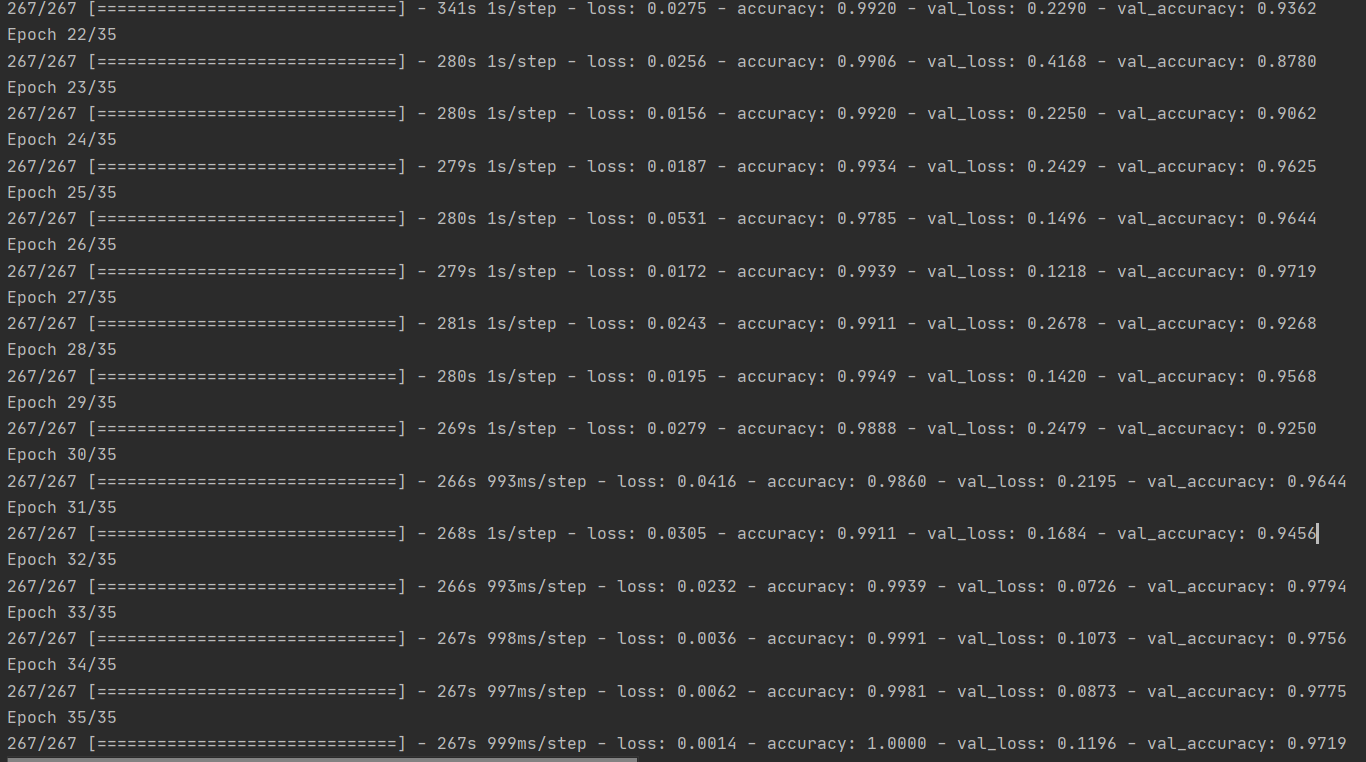


**Run № 4:**

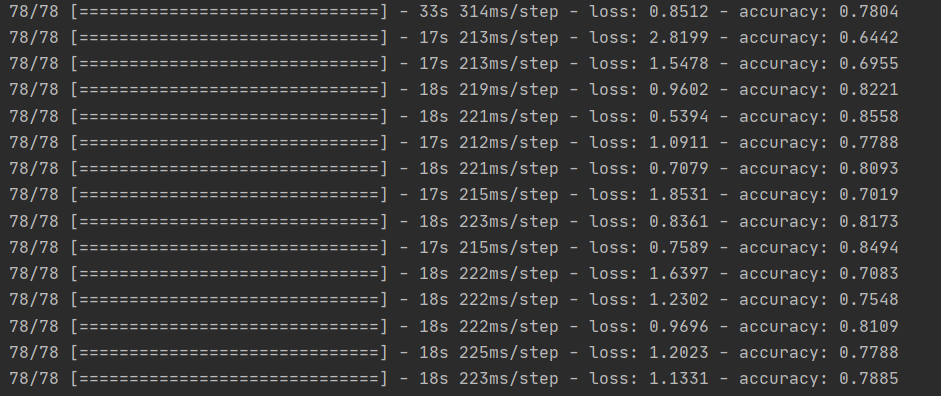
Зареждане на модела от .h5 файл в състояние мрежата- епоха 20 и трениране за още 15 епохи

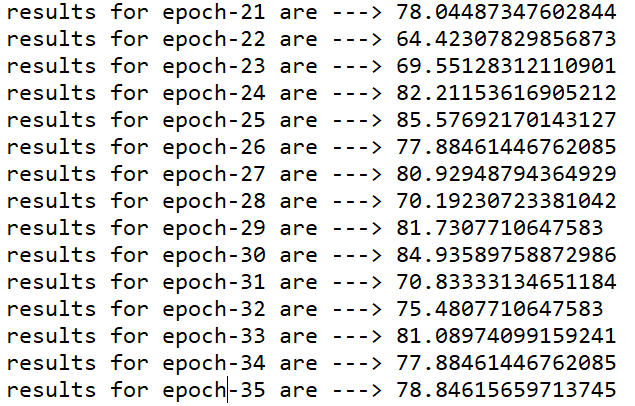
Параметри:

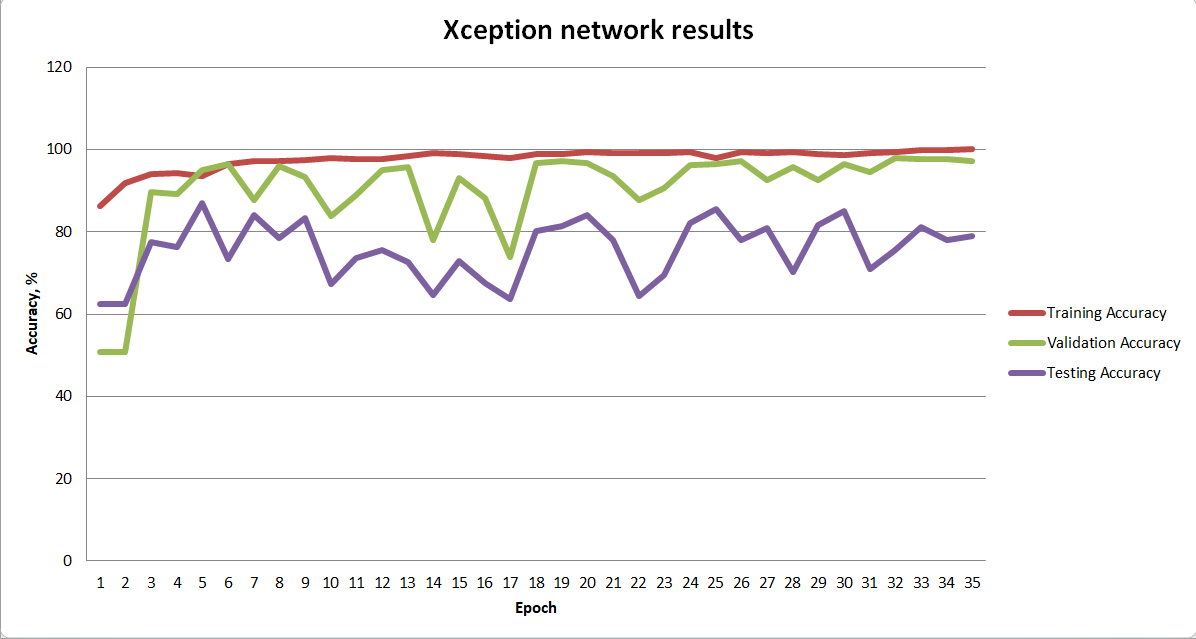
optimizer- Adam(learning\_rate=0.001) ; loss function- binary crossentropy ; **number of epochs – 15(total of 35)**

****

Резултати от тестването :







След епоха 32 се наблюдава насищане на стойността на валидационната точност- около 97% в 4 последователни епохи. Забелязва се и голяма разлика между точността при трениране и тестване на мрежата, което сигнализира за доста вероятен overfitting на модела спрямо данните за трениране.

**Run № 5:**

Зареждане на модела от .h5 файл в състояние мрежата- епоха 35 и трениране за още 10 епохи

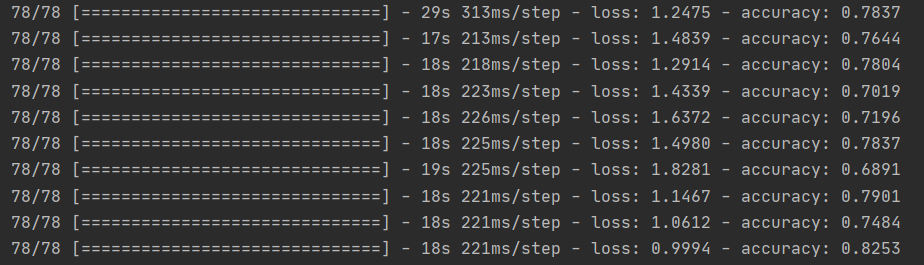
Параметри:

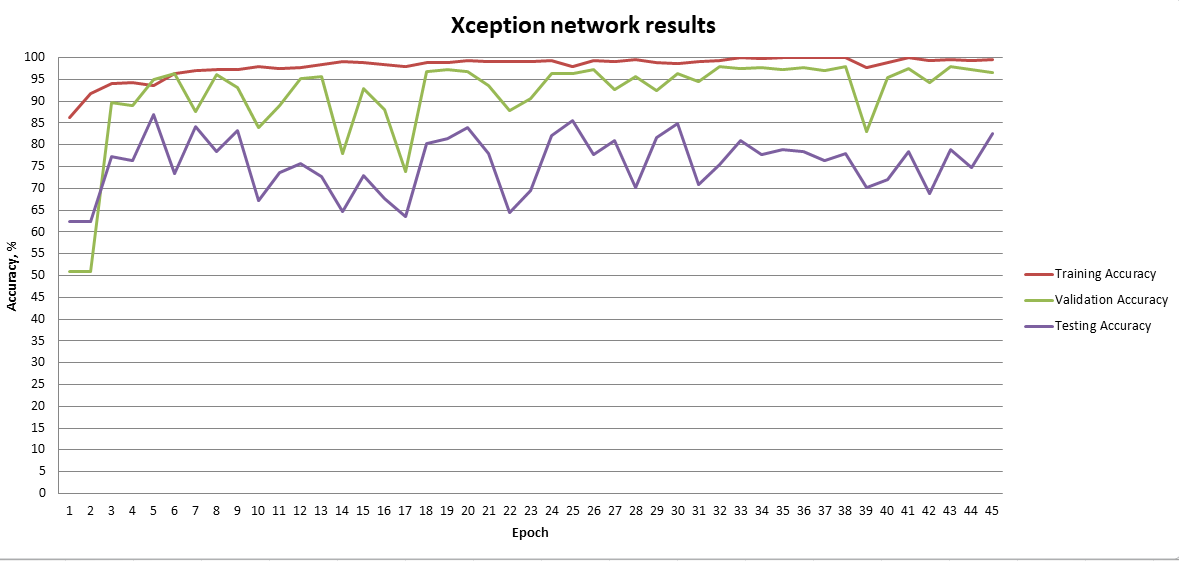
optimizer- Adam(learning\_rate=0.001) ; loss function- binary crossentropy ; **number of epochs – 10(total of 45)**

Резултати при тренирането:



Резултати при тестването:





Наблюдава се и осцилиране на валидационната точност. Възможна причина може да бъде малкият размер на едновременно разглеждани изображения(batch size) – 8. Тренирането на мрежата е извършено без data augmentation. За преодоляване на осцилирането и намаляване на overfitting-a може да бъдат предприети следните мерки:

1. Увеличаване на batch size параметъра(в рамките на разумното и до колкото позволява хардуерът)

2. Промяна на learning rate параметъра с по- малка стойност

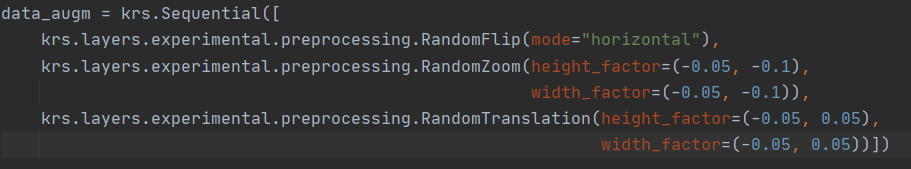
3. Промяна в архитектурата на мрежата.

4. Добавяне на „обогатяване на данните“ - data augmentation

**Опит 2:**

При опит 2 са приложени следните мерки за ограничаване на осцилирането и overfitting-a:

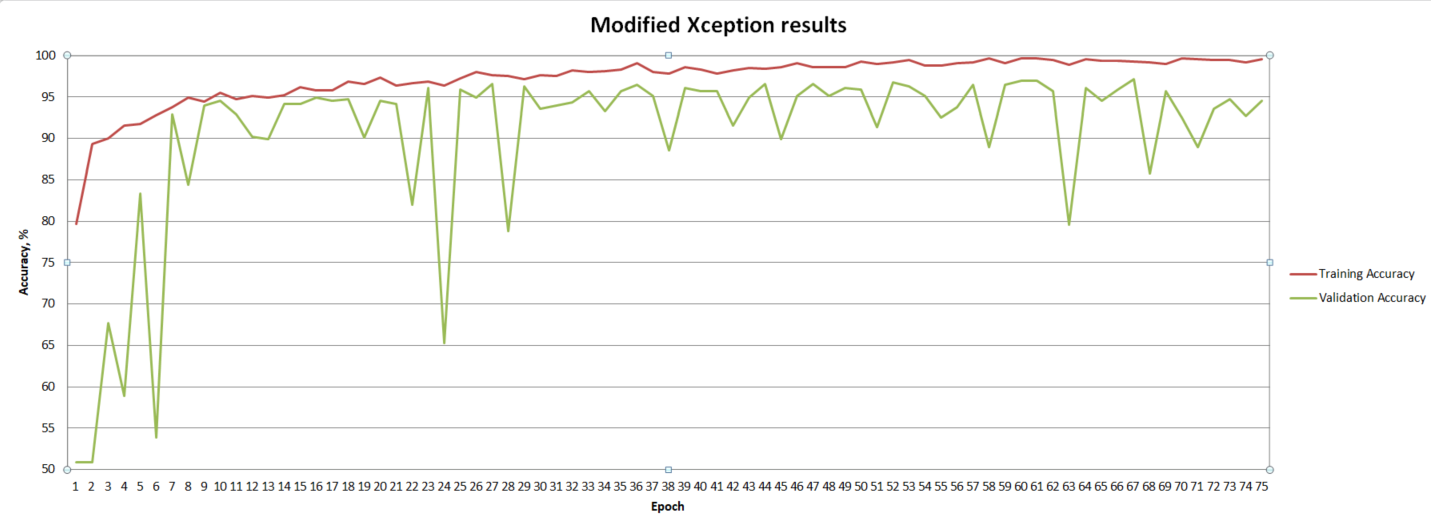
* Добавяне на data augmentation **към архитектурата на невронната мрежа(**обогатяването се извършва „on-line“ върху GPU като слоеве от мрежата**)**

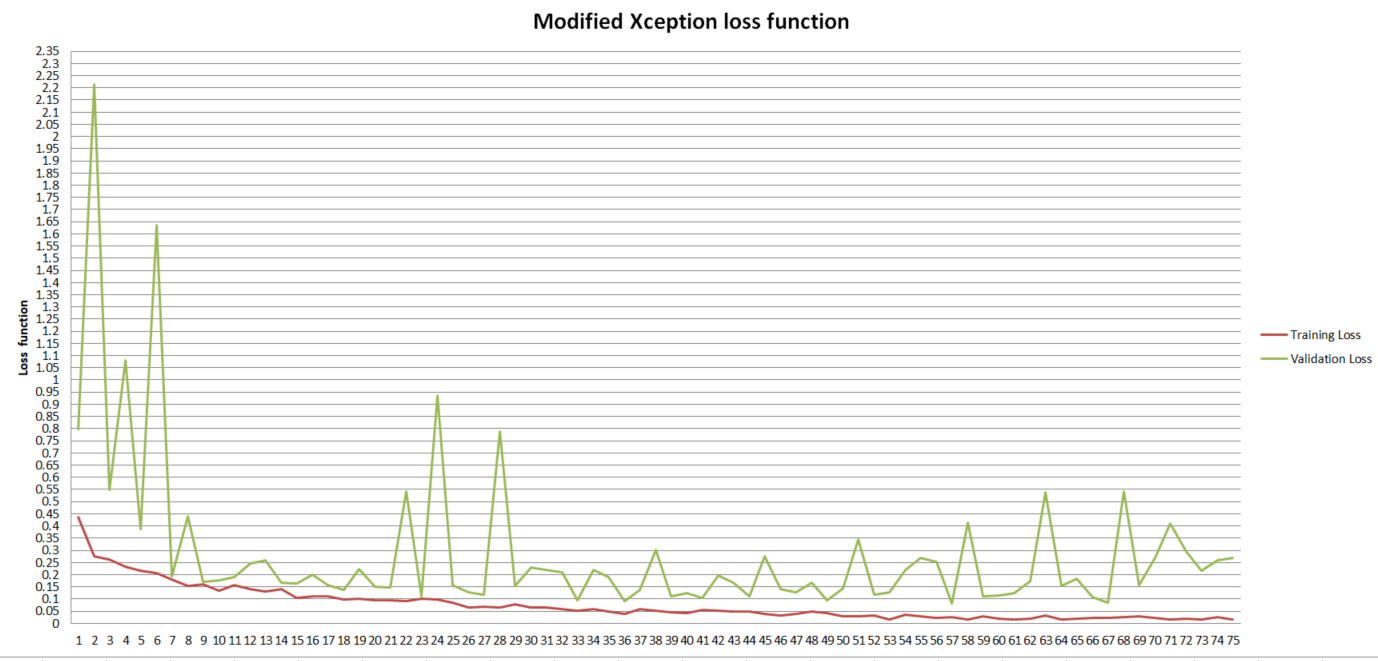


* Промяна на стойността на вероятността на DropOut слоя в мрежата на 0.6 (предишна стойност – 0.5)
* Промяна на хиперпараметъра learning rate на оптимизационния метод Adam на 0.00005(предишна стойност 0.001)
* Стойността на batch size е запазена същата- batch size= 8
* Зададено е отпечатване на следните метрики след всяка епоха – Accuracy, AUC, TP, FP, TN, FN

**Run № 1**

45 епохи





Отново се наблюдава осцилиране в резултатите на точността, но то е значително по- малко.

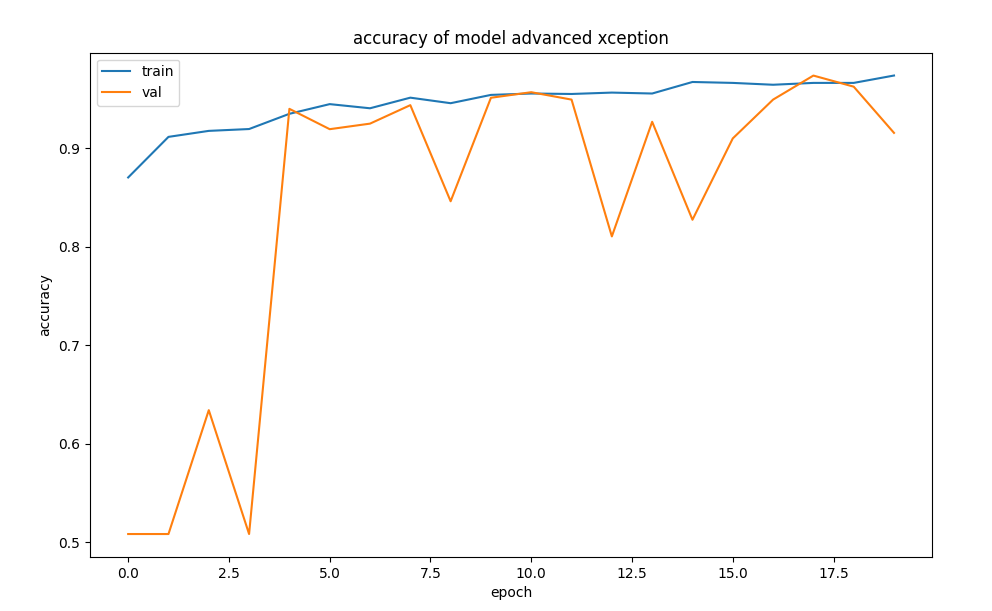
**Run № 2:**

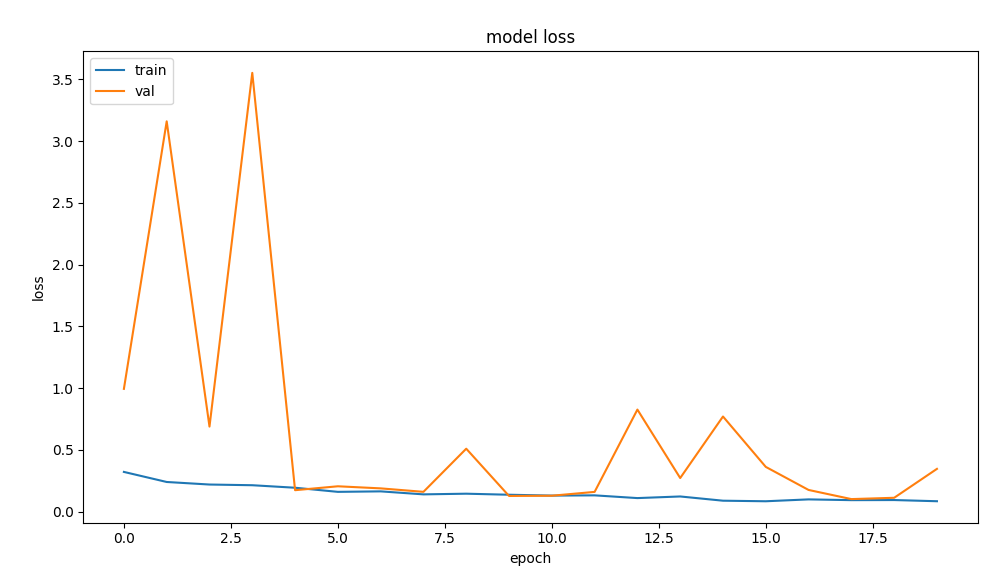
Използвана е същата архитектура на НМ, но с друг оптимизиращ алгоритъм- RMSProp:

model.compile(  
 optimizer=keras.optimizers.RMSprop(learning\_rate=0.0001),  
 loss="binary\_crossentropy",  
 metrics=["accuracy", "AUC",  
 keras.metrics.TruePositives(),  
 keras.metrics.FalsePositives(),  
 keras.metrics.TrueNegatives(),  
 keras.metrics.FalseNegatives()]  
)

Тренирането е извършено за 20 епохи.

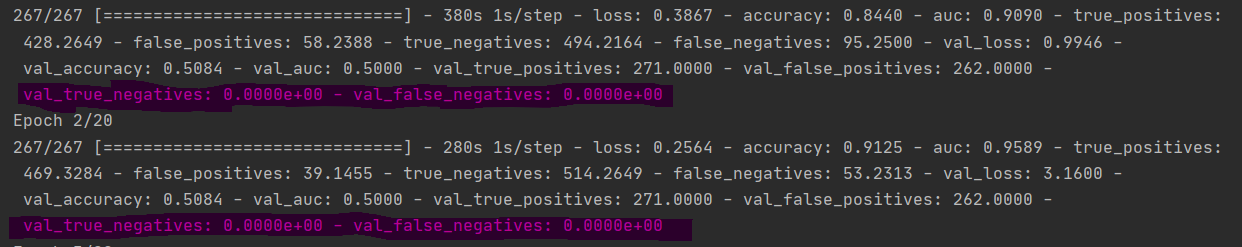
Резултати:



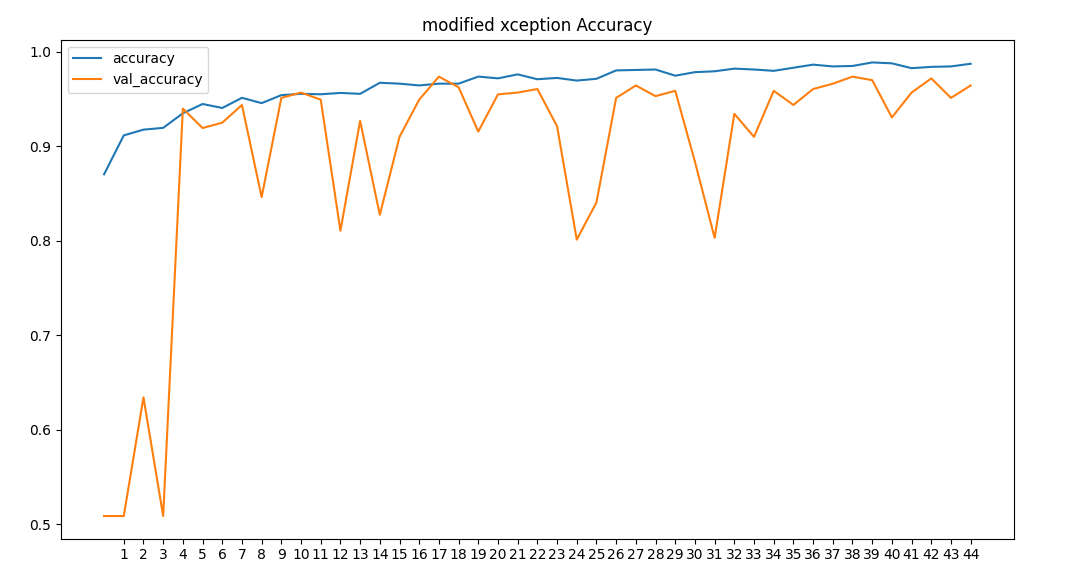


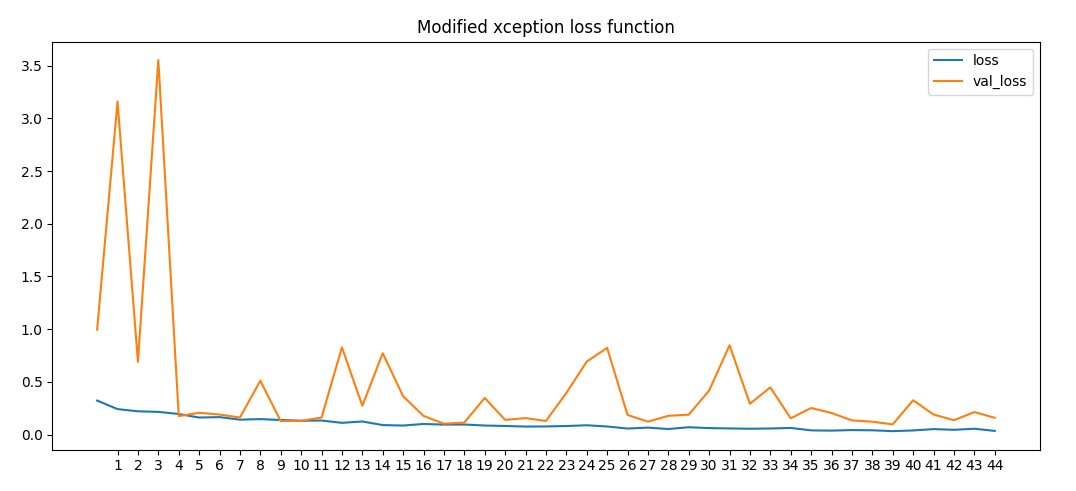
Осцилирането отново е по- малко спрямо Опит 1, но са нужни още данни(още трениране на мрежата), за да се направи заключение и да се открие ясна тенденция.

Интересен факт е, че в първите 2 епохи, невронната мрежа класифицира всички изображения към един единствен клас:



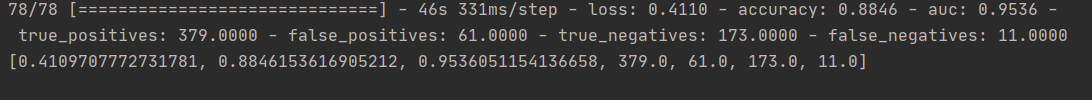
Следва трениране на мрежата за още 25 епохи(общо 45)





От графиките по-горе се наблюдава намаляване на осцилирането и засилване на монотонността след епоха 33. От графиката личи един потенциално най-добър(до момента) модел- от епоха 40.

За него след оценка чрез **тестовия** dataset(около 600 изображения) се получава:



От резултатите се вижда, че достигната точност **е 88.46%,** но стойността на loss функцията остава сравнително висока – 0.411.

Могат да бъдат изчислени и :

Чувствителността - True Positive Rate(TPR) = TP/ TP +FN= 0.972

Прецизността – Positive Predictive Value(PPV) = TP/TP+FP = 0.8614

В конкретния случай се разглеждат медицински изображения. Следователно, според мен e по-важно да се минимизира броя на False Negative класификациите(т.нар. Type 2 error), защото представлява по-голям риск.

**Опит 3.**

В този опит е използвана методиката „трансферирано обучение“(transfer learning). На база наблюдаваното осцилиране от предния опит, в текущия ще бъде използвана резолюция на изображенията- 128х128. Това позволява увеличаване на batch size параметъра(броят едновременно подавани изображения на мрежата) на 32.

**Run 1:**

Използвана е вградената в Tensorflow имплементация на невронната мрежа VGG16, в която са заредени теглата за ImageNet дейтасета. Тяхната промяна е забранена чрез trainable=false по време на трениране върху изображенията на белите дробове. Изображенията в този опит са заредени в RGB формат, защото използваната VGG имплементация го изисква. От архитектурата са махнати последните 3 слоя(пълносвързани и изходният слой с категориите) и са добавени нови, подходящи за текущо обработвания сет:

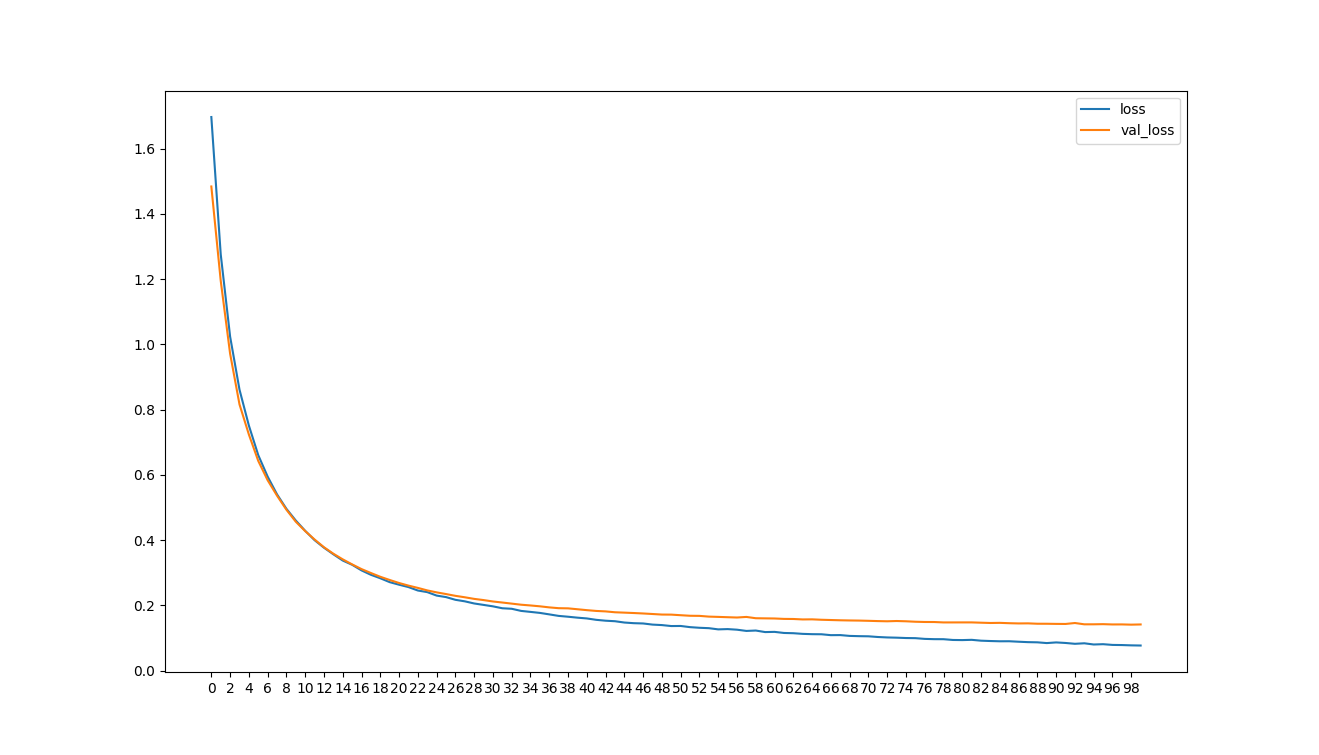
* GlobalAvaragePooling2D- пулинг слой
* Dense(1,activation=’sigmoid’)- изходен слой отразяващ резултата от мрежата в рамките на 1 неврон, чрез който се извършва класификацията. Като активираща функция е зададена сигмоидната, удобно ограничаваща резултата в интервала [0,1].

Извършено е трениране за 100 епохи, със следните параметри :

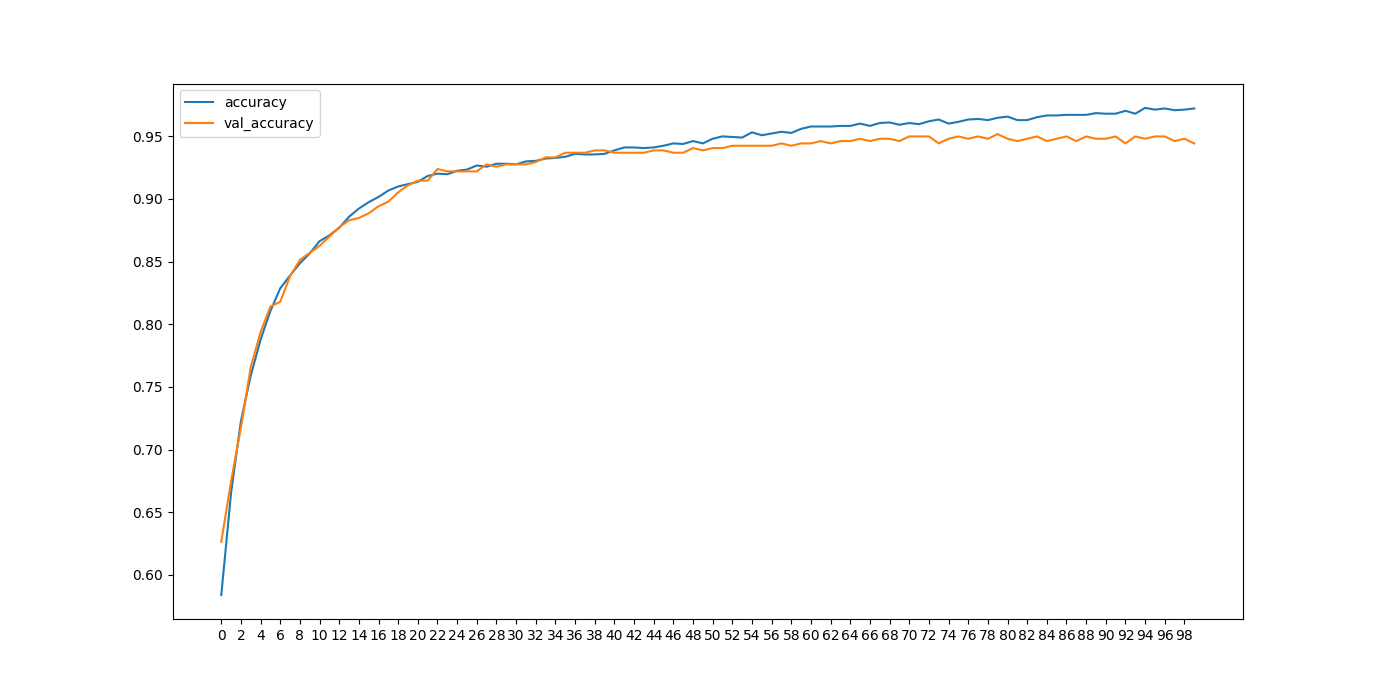
optimizer=keras.optimizers.Adam(learning\_rate=0.0001),  
loss="binary\_crossentropy"

При което се получават следните резултати:

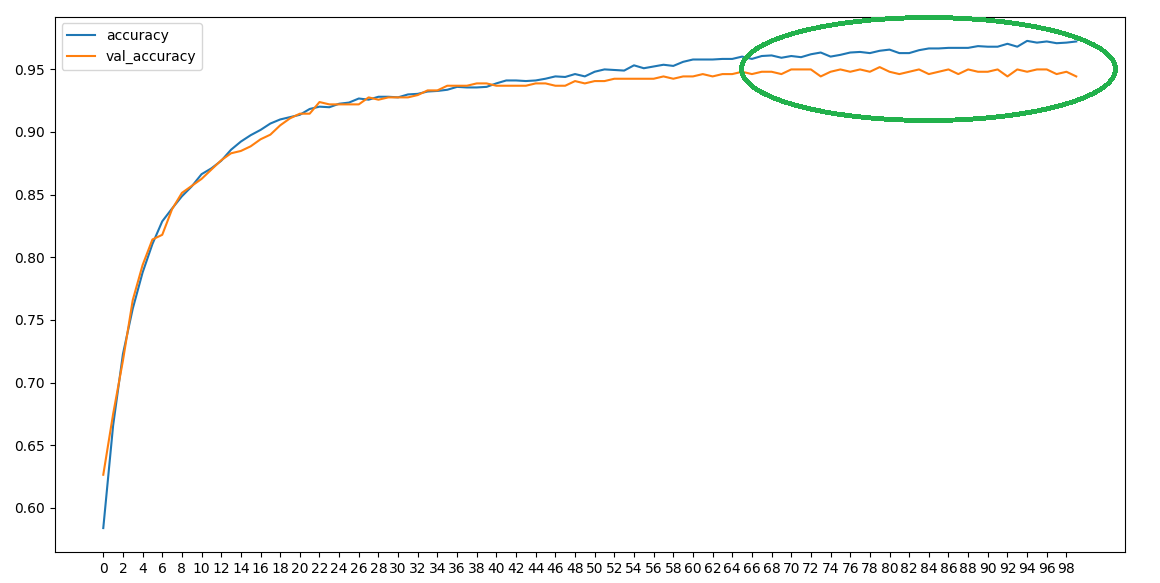
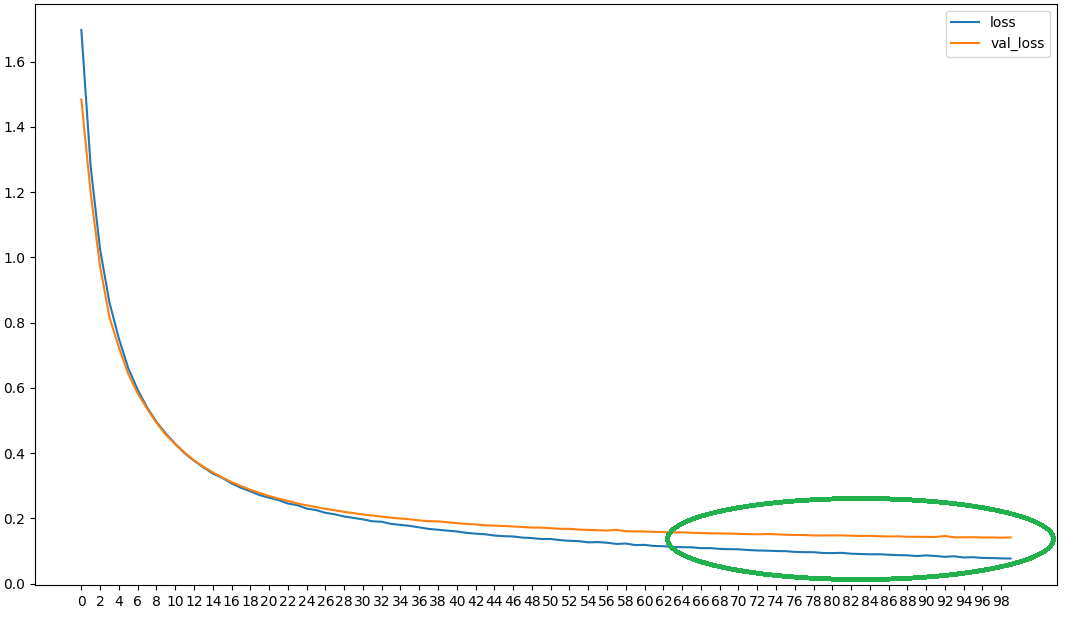
* За ценовата функция



* За точността



Наблюдават се много по-гладки криви на ценовата функция и точността, защото се изчисляват осреднените стойности за всеки batch от изображения, а той е с по-голям размер- 32, спрямо 8 в предишните опити. От графиките се вижда насищане стойностите за ценовата функция и точността след епоха 64-65- отбелязано със зелена елипса:



Може да се заключи, че стойностите на теглата в епоха №65 са оптимални за текущата архитектура. В следващите епохи се вижда повишаване на точността при тренирането, но насищане(на места и спад) във валидационната точност- индикация за пренагаждане на модела спрямо данните за трениране. Overfitting-ът в случая може да се определи като слаб, защото разликата между валидационната и трениращата точност не е голяма- от порядъка на 5%. По-нататъшно трениране(след епоха 100) няма да доведе до по-добри резултати, а напротив- моделът няма да може да генерализира достатъчно добре и да класифицира правилно нови, невиждани до сега данни.

**Run №2**

В този опит към архитектурата на VGG16 от предния опит ще бъдат добавени следните модификации:

* **Data Augmentation слой-** произволни модификации върху входните изображения
* **Dropout слой** с параметър вероятност 0.3 – за да се избегне пренагаждането на модела спрямо данните за трениране и да успява да генерализира по- добре
* **Early stopping-** използване на техниката за навременно спиране на тренирането за да се избегне **overfitting.** Като критерий за прекратяване на обучението е зададен – **липса на промяна в стойността на валидационната ценова функция в 3 последователни епохи.**

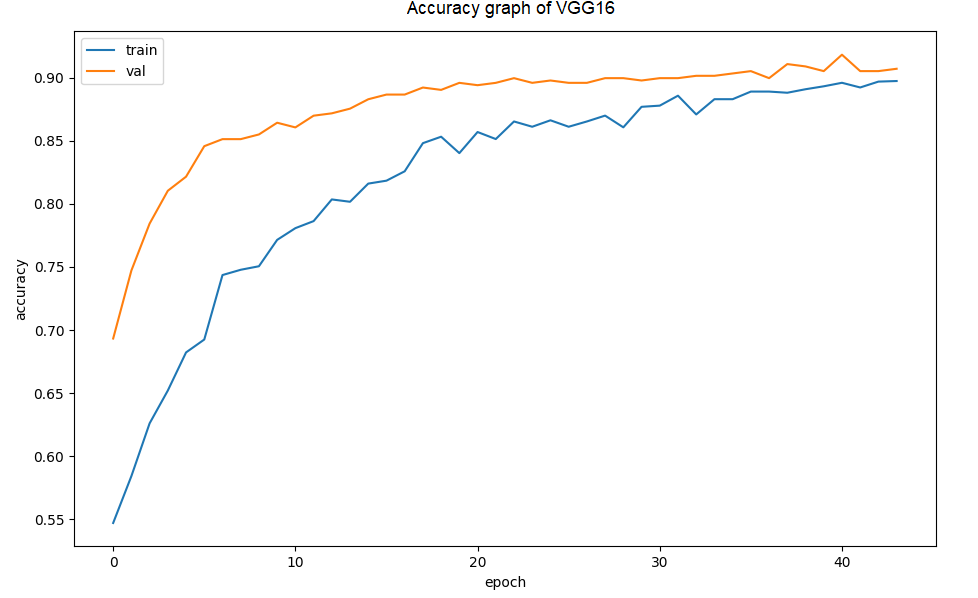
Код на създаване на модела:

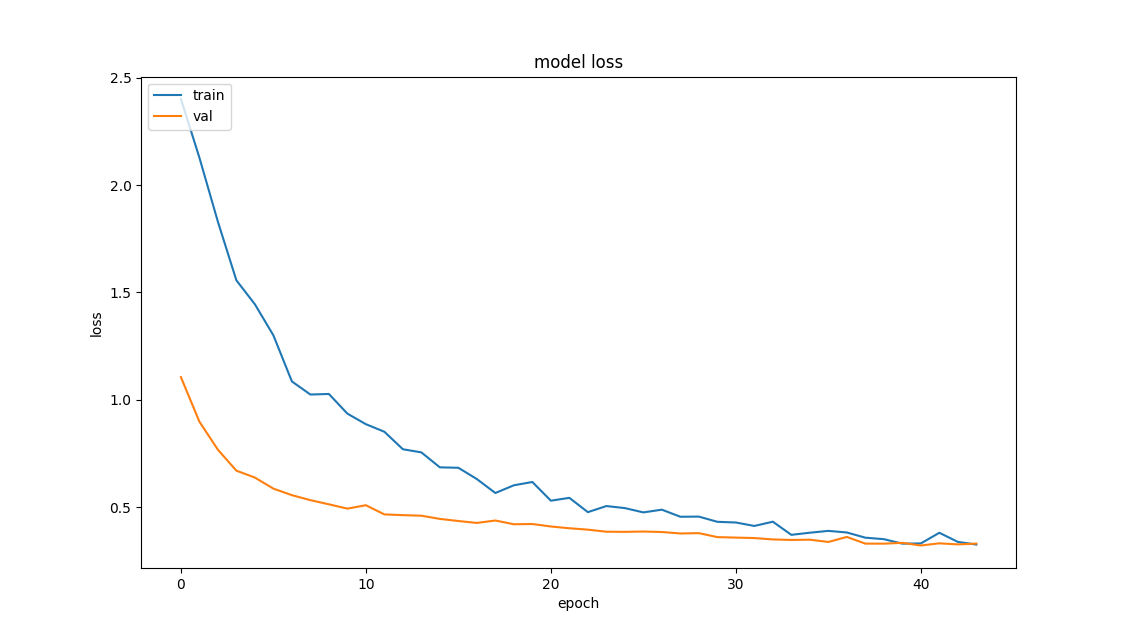
def model\_create\_vgg16(input\_shape):  
 base\_mdl = krs.applications.vgg16.VGG16(include\_top=False, Input\_shape=input\_shape, weights='imagenet')  
 base\_mdl.trainable = False  
 inputs = krs.Input(input\_shape)  
 inp = data\_augm(inputs)  
 inp = krs.applications.vgg16.preprocess\_input(inp)  
 val = base\_mdl(inp, training=False)  
 val = krs.layers.GlobalAveragePooling2D()(val)  
 val= krs.layers.Dropout(0.3)(val)  
 outputs = krs.layers.Dense(1, activation='sigmoid')(val)  
  
 return krs.Model(inputs, outputs)

Код на компилация и трениране на модела:

def train\_vgg():  
 training = datalib.load\_dataset(TRAINING\_PATH)  
 val\_ds = datalib.load\_validation(TRAINING\_PATH)  
 model = mdl.model\_create\_vgg16(input\_shape=(128, 128, 3))  
 epochs = 50  
 callbacks = [  
 keras.callbacks.ModelCheckpoint("vgg2/save\_at\_{epoch}.h5", save\_best\_only=True),  
 # keras.callbacks.CSVLogger(filename="xception\_log.csv", separator=',', append=True)  
 keras.callbacks.EarlyStopping(monitor='val\_loss', patience=3, mode='auto')  
 ]  
 model.compile(  
 optimizer=keras.optimizers.Adam(learning\_rate=0.0001),  
 loss="binary\_crossentropy",  
 metrics=["accuracy", "AUC",  
 keras.metrics.TruePositives(),  
 keras.metrics.FalsePositives(),  
 keras.metrics.TrueNegatives(),  
 keras.metrics.FalseNegatives()]  
 )  
 history = model.fit(  
 training, epochs=epochs, callbacks=callbacks, validation\_data=val\_ds,  
 )

Резултати от тренирането :

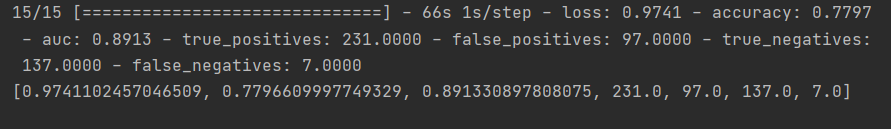




От представените графики на резултатите за стойността на ценовата функция и точността се вижда, че техниката за ранно спиране на тренирането е прекратила процеса в епоха 43(не е имало подобрение в епохите 41,42,43). Наблюдава се и минимално осцилиране, което най-вероятно се дължи на добавеното обогатяване на данните, внасящо по-голямо разнообразие в изображенията и респективно „случайност“. Интерес буди и фактът, че валидационната точност е по- висока от точността на трениране, а стойността на ценовата функция при валидацията е по-ниска от тази при трениране(т.е. моделът се справя по- добре при валидацията от колкото при тренирането). Това е така, защото архитектурата притежава изходен Dropout слой с определена вероятност, който е активен **само по време на тренирането.** Изкуствено затрудняваме мрежата по време на тренирането(загубата на информация от неврони), затова е логично и точността/ценовата функция да имат по-ниска/висока стойност.

Оптималните тегла при тренирането са достигнати в епоха 41, следователно тя ще бъде използвана и при оценка с тестовия дейтасет.

Резултати от тестването :



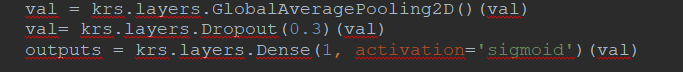
Достигнатата точност при тестовия дейтасет е 77.97%. Въпреки високото ниво сгрешени предположения от мрежата, по-тежкият вид грешка(Type 2 error- фалшиво отрицателни стойности) е доста по-малка спрямо Type 1 error- фалшиво положителните:

* False positives: 97 samples (Type 1 Error)
* False negative: 7 samples (Type 2 Error)

За конкретния проблем, по-голяма заплаха би представлявала именно грешка тип 2.

**Run №3**

В предишния run бяха премахнати пълно свързаните крайни слоеве на традиционната VGG16 архитектура и бяха добавени GlobalAvgPooling слой и веднага след него – класификатора със сигмоидната функция:



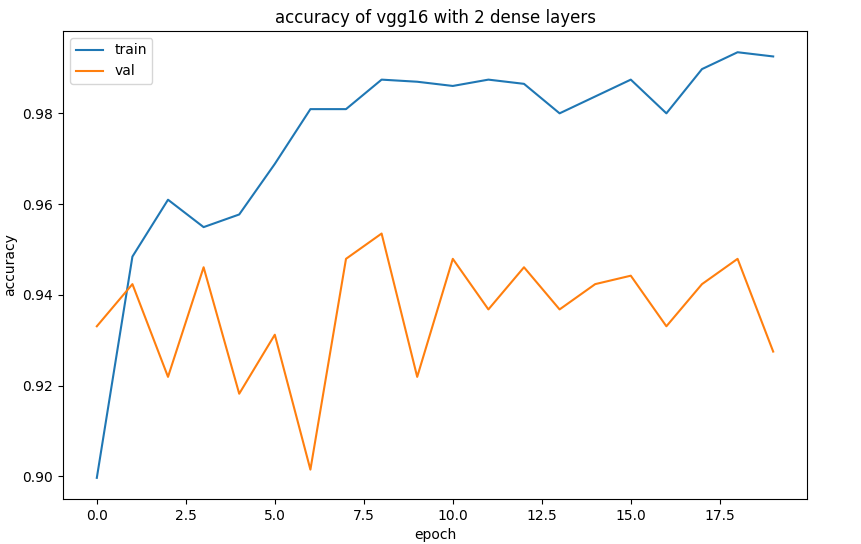
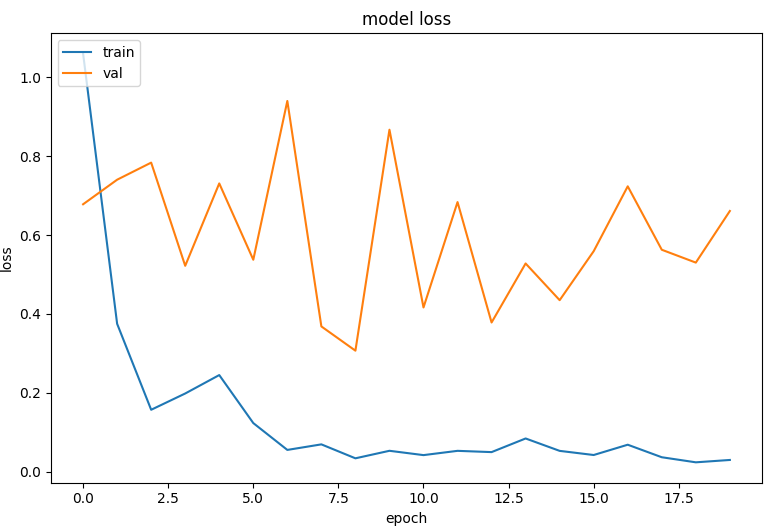
Както се вижда от графиките на предното пускане, точността, която се достига при трениране/валидация е от порядъка на 80-90%, а тази при тестване с тестовия дейтасет- дори по-ниска. Много вероятно е изходните слоеве да не успяват да усвоят достатъчно добре специфичността на данните. Затова ще бъде направена промяна в крайните слоеве на мрежата- flatten слой и два плътно свързани слоя с ReLu активационна функция преди същинския класификатор(маркирани в син цвят). Така архитектурата придобива следния вид:

def model\_create\_vgg16(input\_shape):  
 base\_mdl = krs.applications.vgg16.VGG16(include\_top=False, input\_shape=input\_shape, weights='imagenet')  
 base\_mdl.summary()  
 base\_mdl.trainable = False  
 inputs = krs.Input(input\_shape)  
 inp = data\_augm(inputs)  
 inp = krs.applications.vgg16.preprocess\_input(inp)  
 val = base\_mdl(inp, training=False)  
 val = krs.layers.Flatten()(val)  
 val = krs.layers.Dense(2048, activation="relu")(val)  
 val = krs.layers.Dense(2048, activation="relu")(val)  
 outputs = krs.layers.Dense(1, activation='sigmoid')(val)  
 return krs.Model(inputs, outputs)

За първоначалното трениране няма да бъде ползван dropout слой. Параметри на тренирането :

* Използван оптимизационен метод: Adam
* Брой епохи: 20
* Learning rate: 0.0001
* Ценова функция: binary crossentropy

**Резултати**

****

В графиките на резултатите отново се наблюдава леко осцилиране(малко по-силно изразено във валидационната част), но доста по-високи нива на достигната точност спрямо предишния опит. Въпреки това, отново се вижда разлика между валидационна точност и точност за трениране от порядъка на 6-7%, която предразполага използването на някакъв вид регуляризация(например въвеждането на dropout слой).

**Learning Rate Scheduler**

Използването на по-гол0ям batch size(32) чувствително успя да намали осцилирането, но не и да го премахне. Друга възможна причина за осцилирането може да бъде learning rate(LR) хиперпараметърът - да има прекалено висока стойност за текущия разглеждан проблем и да „прескача“ оптималното решение. За справяне с това явление в следващото трениране ще бъде използвана и техниката **learning rate schedule,** при която LR параметърът ще бъде намаляван прогресивно във времето. Това ще спомогне по-лесното намиране на оптималното решение, защото теглата ще се обновяват с по- малки стъпки. Това, разбира се, крие и риск- възможно е засядане в локален минимум, ако LR параметърът е прекалено малък.

Tensorflow изисква задаването на функция за динамично определяне на learning rate. Съществуват вече написани такива в библиотеката, но в текущия опит ще бъде използвана собствена имплементация на шедулиращата функция със следния код:

MIN\_LEARNING\_VALUE = 0.000000001

def lr\_scheduler(epoch, lr):  
 if epoch < 5 or lr < MIN\_LEARNING\_VALUE:  
 return lr  
 elif 5 <= epoch < 15:  
 return lr\*0.8  
 else:  
 return lr\*0.5

В първите 5 епохи няма да има промяна в LR параметъра, от епоха 5 до 15 той ще бъде намаляван с фактор 0.8, а след 15 епоха- наполовина.

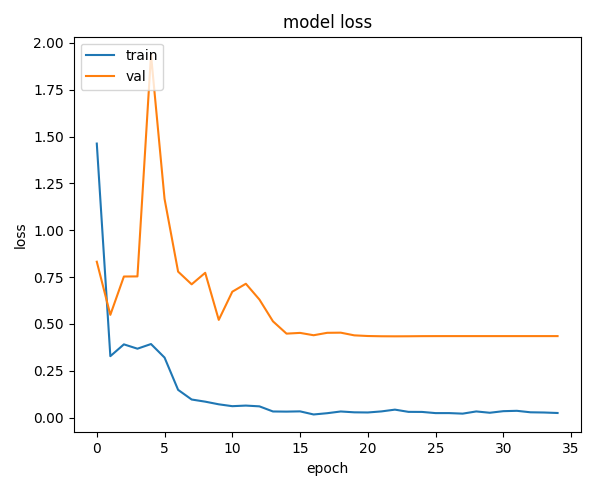
**Dropout**

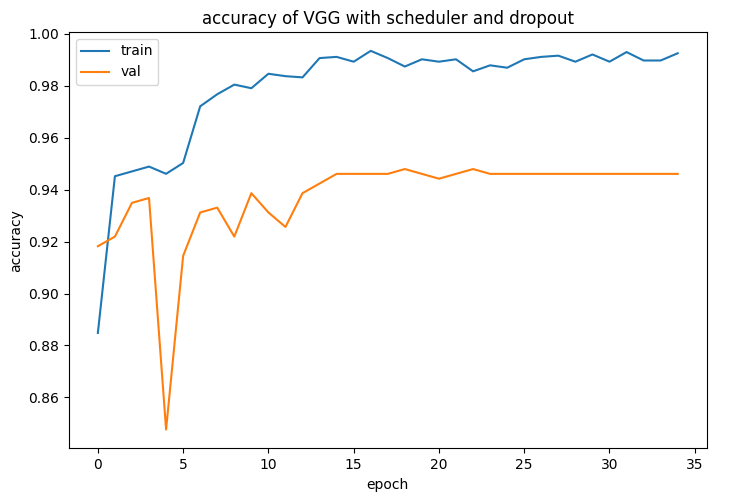
Добавяне на dropout слой като мярка за регуляризация и респективно намаляване на пренагаждането на модела:

val = krs.layers.Flatten()(val)  
val = krs.layers.Dense(2048, activation="relu")(val)  
val = krs.layers.Dropout(0.3, seed=1333)(val)  
val = krs.layers.Dense(2048, activation="relu")(val)

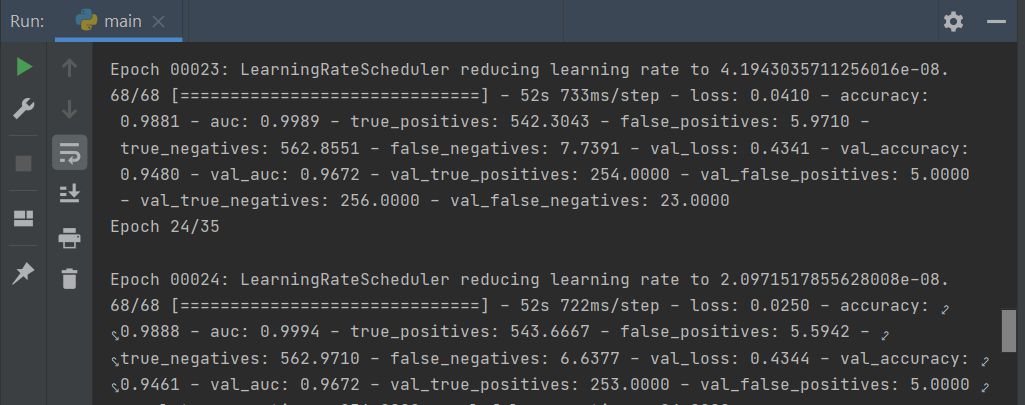
Поставен е dropout слой с вероятност 0.3 между двата плътносвързани слоя в края на мрежата. Като параметър е подаден и seed, за да може да бъде пресъздадено същото действие на слоя в следващи опити.

**Резултати:**





На графиките се забелязва стагнация в стойностите на точността и ценовата функция след епоха 23. В този момент скоростта на обучение е изключително ниска:



Learning rate ~ 4,2 .10-8

Както бе споменато по-горе, прекалено ниският Learning rate може да доведе до засядане в локален минимум. Такова засядане се наблюдава и в текущия опит. Кривата на трениране показва повече осцилиране отколкото тази на валидация, защото слоевете за обогатяване на данните и Dropout са активни само по време на тренирането, внасяйки случаен елемент и по-голямо разнообразие. Отново се забелязва разлика между валидационни и тренирани резултати- например при точността - δ ~ 4%.

На база резултатите ще бъдат направени следните промени:

1. Модифициране на Learning rate schedule функцията- по-плавно намаляване на стойността и по- висока стойност на границата

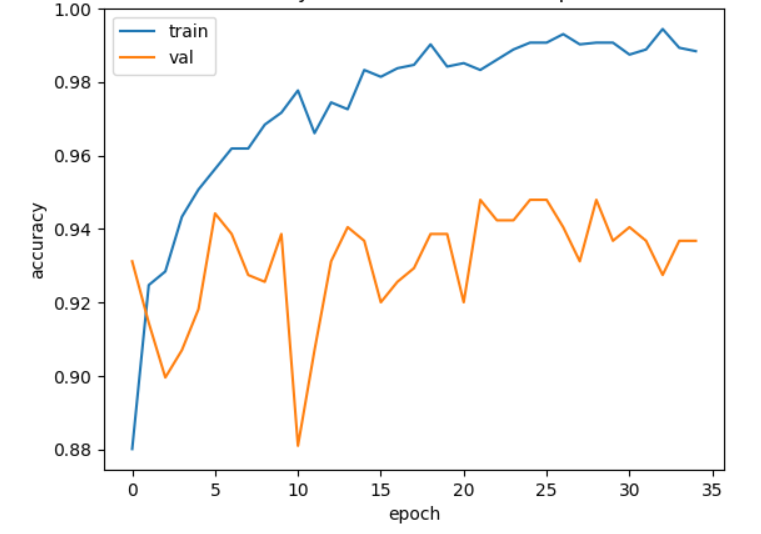
MIN\_LEARNING\_VALUE = 0.0000001

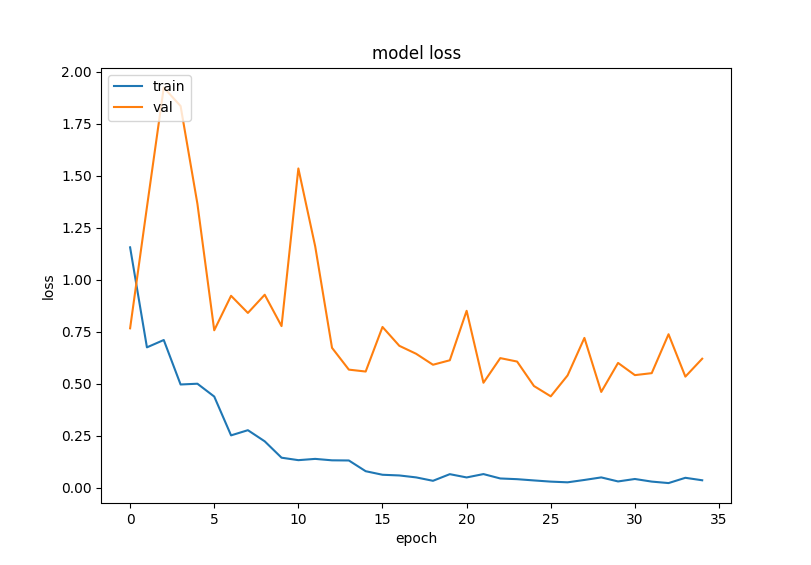
def lr\_scheduler(epoch, lr):  
 if epoch < 5 or lr < MIN\_LEARNING\_VALUE:  
 return lr  
 elif 5 <= epoch < 15:  
 return lr\*0.9  
 else:  
 return lr\*0.97

1. Увеличаване на dropout вероятността от 0.3 на 0.5

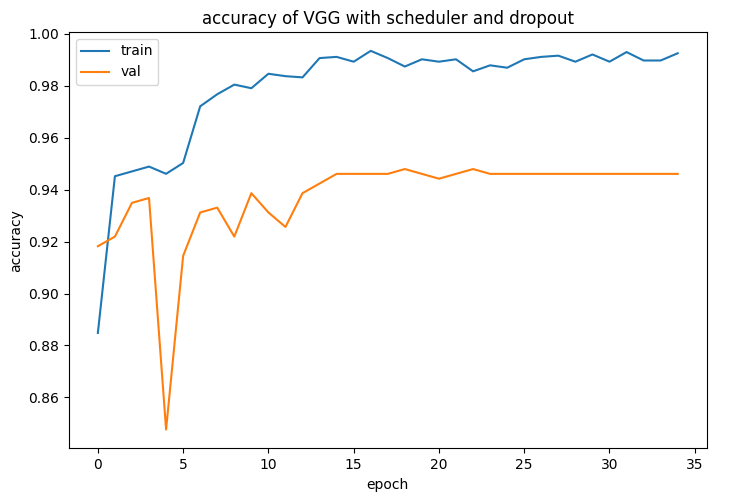
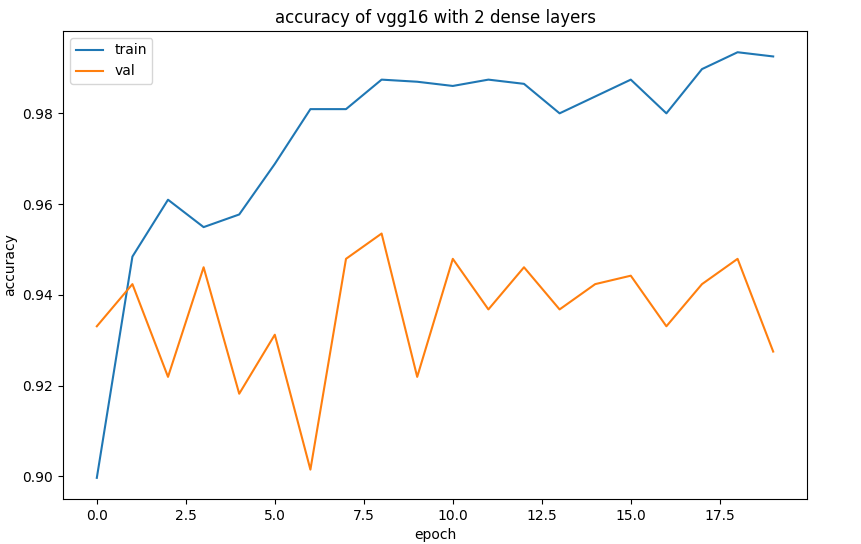
al = krs.layers.Flatten()(val)  
val = krs.layers.Dense(2048, activation="relu")(val)  
val = krs.layers.Dropout(0.5, seed=1333)(val)  
val = krs.layers.Dense(2048, activation="relu")(val)

**Резултати:**

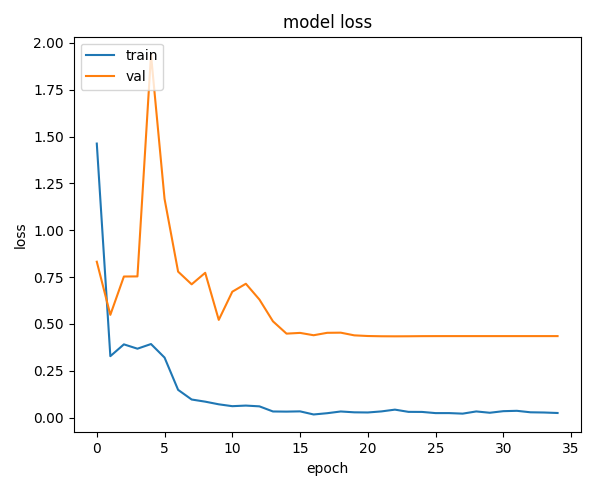
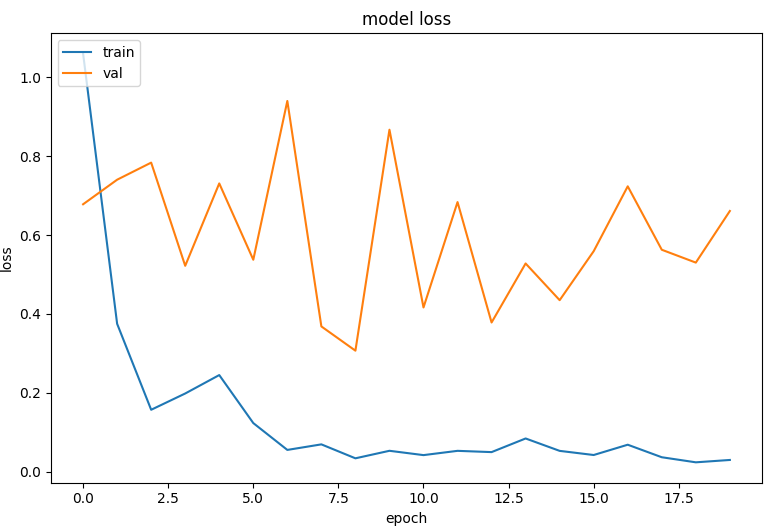
****

****

На база горните графики може да се заключи, че няма особена полза от използване на learning rate scheduler методиката в за конкретния проблем съвместно с адаптивния оптимизатор Adam. Този оптимизационен алгоритъм принадлежи на семейството на т.нар адаптивни оптимизатори(Adaptive Optimizers), които динамично изчисляват и задават различен learning rate за различните параметри от мрежата и не се нуждаят непременно от ръчно намаляване на LR. Шедулирането на LR би било по- полезно при по-старите оптимизационни методи, като например стохастичното градиентно спускане(stochastic gradient descent SGD), използващо иначе постоянна стойност за LR. Ако поставим графиките за **точността** в опитите **с и без динамично намаляване** на коефициента на обучение:

****

Вижда се, че и в двата случая оптималната достигната точност е около 94%, дори и ако се разглеждат графиките рамките на по-краткия опит(20 епохи). Този факт засилва твърдението, че за текущият проблем, използването на Scheduler не носи особени ползи. Подобен извод може да се направи и при наблюдаване на графиките на ценовата функция. Стойността се запазва около 0.5(лява графика с scheduler, дясна без):

****

**Опит 4.**

След направените заключения за динамичния learning rate в контекста на адаптивните оптимизатори, в този Run ще бъдат разгледано влиянието на Епсилон (ϵ) константата за числова стабилност в Adam оптимизатора. Както се вижда от формулата по-долу, тя е поставена в знаменател за да гарантира, че няма да се случи деление на 0. Нула в знаменателя може да се получи при така наречените изчезващи градиенти, при които се получават изключително ниски стойности за градиента. Понякога те дори може да бъдат закръглени към 0 от някои компилатори и системи. В оригиналната публикация на **Kingma and Ba**, тя е обозначена като „ϵ черта“:



, където с θ се отбелязва конкретен параметър на целевата функция.

Aвторите на публикацията предлагат стойност на ϵ= 1. 10-7 , като подходяща в голяма част от случаите. От друга страна, според документацията на Keras библиотеката за Adam алгоритъма, тази стойност не е добре да се използва като стойност по подразбиране, а дори да се приеме за хиперпараметър. Документацията дори дава примерни подходящи стойности за епсилон константата в контекста на Inception мрежата ϵ=1 или ϵ=0.1. Именно затова, ще бъдат тествани различни стойности на епсилон параметъра. Ще бъде използвана VGG16 архитектурата от предния опит

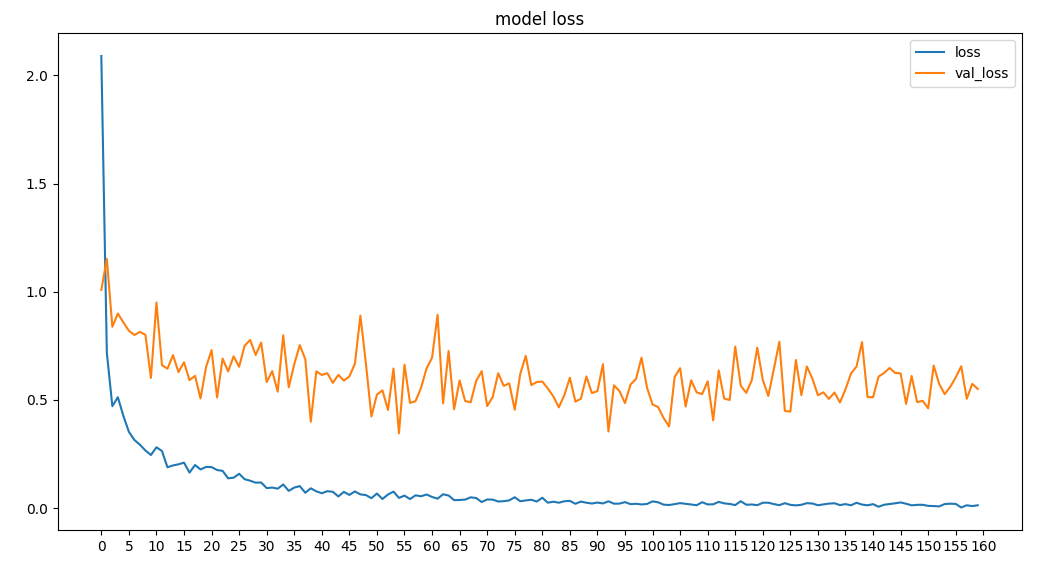
**Run №1:**

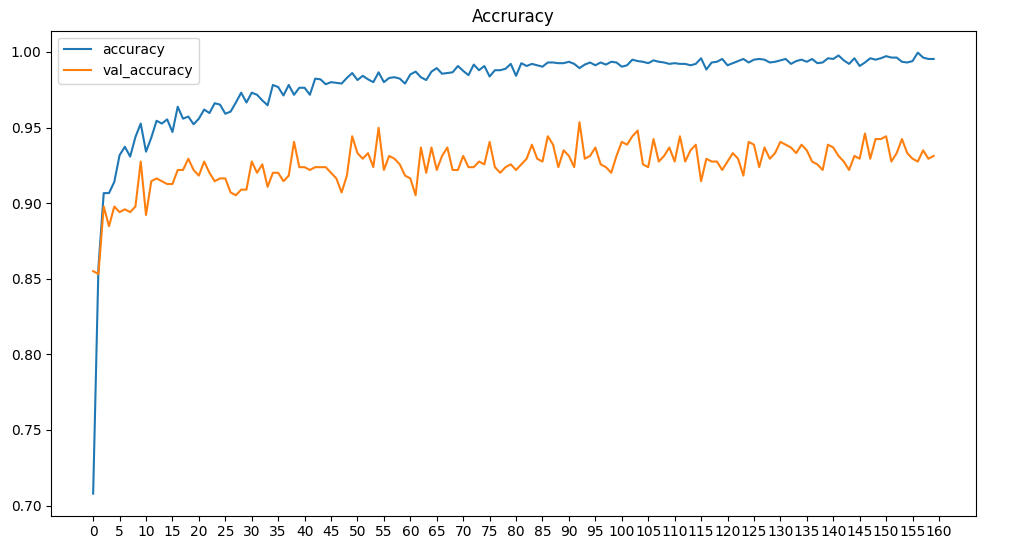
Параметри на текущото трениране:

Optimizer: **Adam**

Learning rate: 0.00001

ϵ = 0.001





Най-добри резултати са достигнати при епоха 55. Наблюдава се повишено осцилиране на валидационната крива спрямо тази на трениране, което е в рамките на допустимото. Амплитудата на осцилиране също е в рамките на допустимото- около 4%.

**Run №2:**

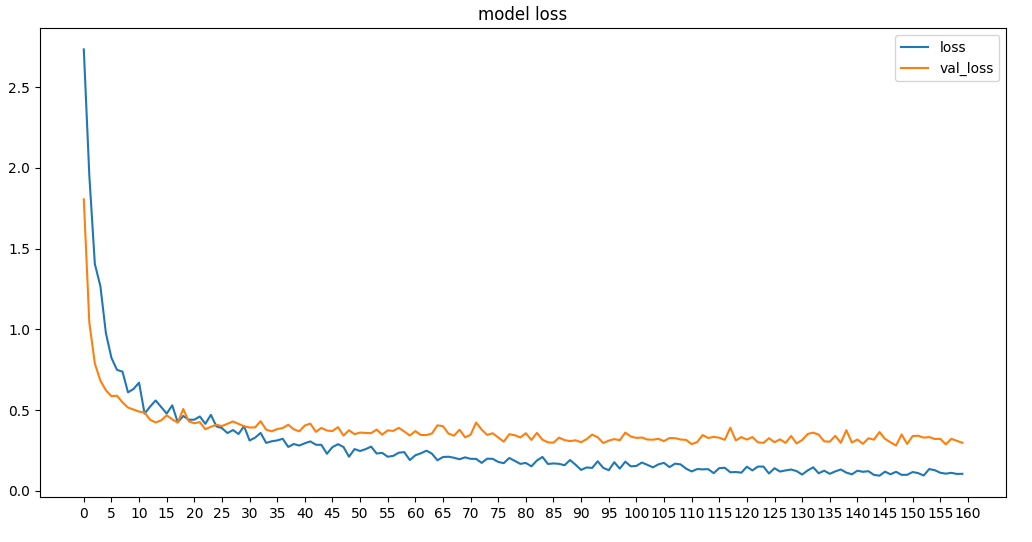
Параметри на текущото трениране:

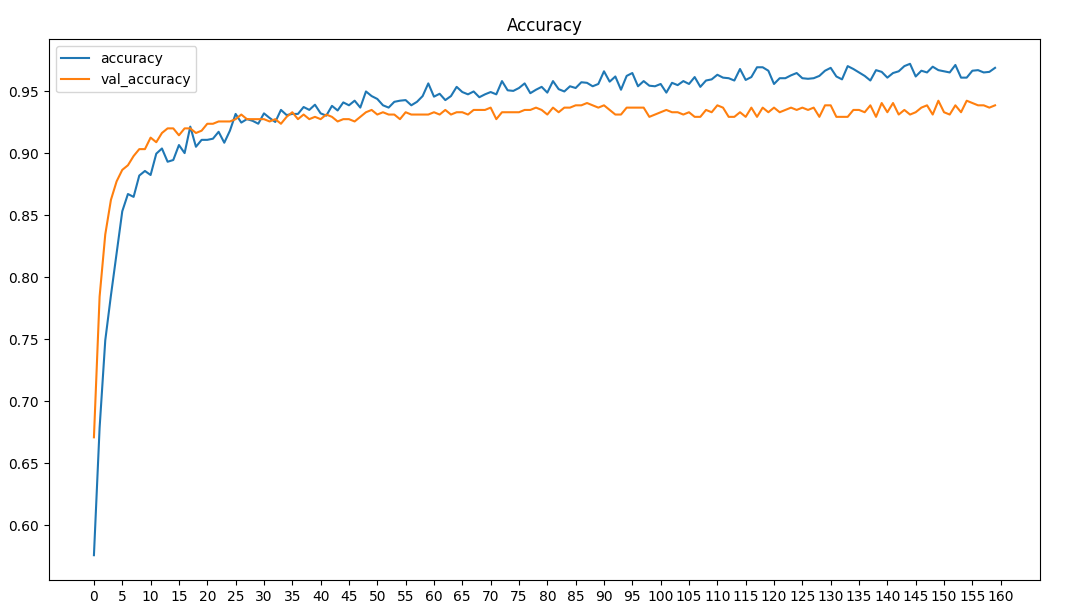
Optimizer: **Adam**

Learning rate: 0.00001

ϵ = 0.1

**optimizer=keras.optimizers.Adam(learning\_rate=0.00001, epsilon=0.1),**





От графиките се вижда се вижда значително намалено осцилиране спрямо тренирането с ϵ = 0.001. Както личи от формулата за обновяване на теглата по-горе, епсилон константата е в знаменател- т.е по-голяма стойност на епсилон би довела до по-малки по големина обновявания на теглата, или грубо казано - по-бавно обучение. В този случай и максималната амплитуда в стойността на валидационната точност е в рамките на 1 %.

Сравнителна таблица :

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Архитектура  Х-ки | Max амплитуда на вал.точност | Max вал. точност | Max разлка между вал. и трен. точност | Min вал. ст-т на ценовата ф-я | епоха | Точност при тестване |
| VGG16(ϵ = 0.001) | ~5 % | 95,35 % | ~8 % | 0.3456 | 93,55 | 87,29 % |
| VGG16(ϵ = 0.1) | ~1 % | 94,24 % | ~4 % | 0.2813 | 148,150 | 82,20 % |

Забележка: Max амплитуда на валидационната точност е отчитана след епоха №10, защото в началото на трениращият процес се очакват ниски стойности.

От получените резултати се вижда, че при по-малката стойност на епсилон от двете тествани се достига по-висока точност върху тестовия дейтасет, въпреки по-голямата амплитуда, по- голямата разлика между вал. и тренираща точност и по-голямата стойност на ценовата функция. Следователно може да заключим, че за текущия разглеждан проблем, VGG16 моделът с ϵ = 0.001 генерализира по-добре от този с ϵ = 0.1.

Резултатите от тестването обаче показват доста по-ниска точност при тестване от тази при трениране(около 5% за ϵ = 0.001 и **над** **12%** за ϵ = 0.1). Причините за това могат да бъдат две:

1. **Причина- Overfitting-** това е по-явната причина от двете. Както беше споменато и по-горе, невронната мрежа се стреми прекалено добре да научи данните и не успява да се справи добре с невиждани досега изображения. Въпреки че това е по-явната причина, то тя е по-малко вероятната в случая поради :

* Приложените техники за предотвратяване на пренагаждането на модела- Dropout, Data augmentation
* Тестване с тестовия дейтасет още във втората епоха отново води до разлика между валидационна и тестова точност- пренагаждането на модела няма как да се случи толкова рано в процеса на трениране.

Възможна предпоставка за пренагаждане на модела е прекалено сложната архитектура на мрежата. Затова ще бъде направен малък тест в рамките на 30 епохи с промяна в броя на невроните в плътносвързаните top слоевете на мрежата(от 2048 на 512).

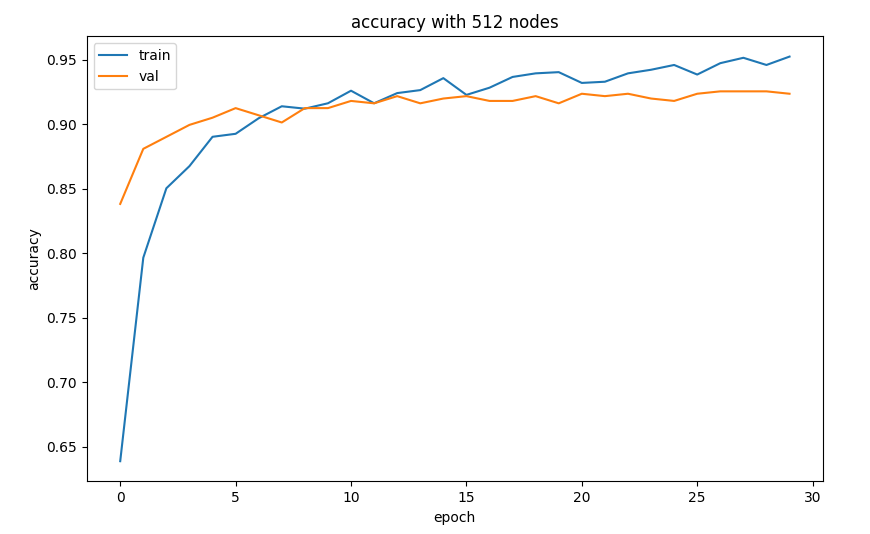
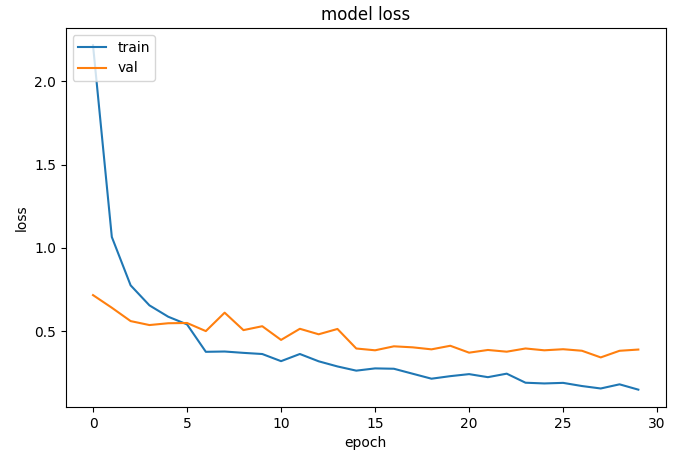
val = krs.layers.Dense(512, activation="relu")(val)

Параметри на тестването:

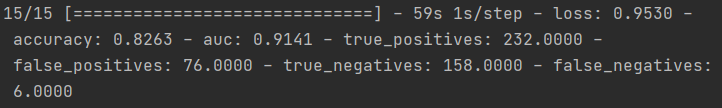
Брой епохи: 30

model.compile(  
 optimizer=keras.optimizers.Adam(learning\_rate=0.00001, epsilon=0.001),  
 loss="binary\_crossentropy",  
 metrics=["accuracy", "AUC",  
 keras.metrics.TruePositives(),  
 keras.metrics.FalsePositives(),  
 keras.metrics.TrueNegatives(),  
 keras.metrics.FalseNegatives()])

Резултати от тренирането:



От графиките може да се приеме, че епоха 15 е оптимална. След нея се забелязва подобряване на трениращата точност/ценова функция, но валидационните резултати нямат видимо подобрение(моделът губи възможността си да генерализира добре). Следователно теглата от епоха 15 ще се използват за оценка на модела върху тестовия сет:



Достигната точност при оценка с тестовия сет е 82,63 % - приблизително с 10% по- ниска от тази при валидация. Това води до изключване на вероятния овърфитинг на модела.

1. **Причина**- Сета за трениране и сета за данни са извадени от различно разпределение- използваните данни за тестване и трениране вероятно не са от едно и също разпределение. При изтеглянето на изображенията от хранилището, те са били предварително разделени в две групи- за тестване и за трениране, но не е дадена подробна информация как се е случило то. За да се провери тази хипотеза, следва да се преработят данните- двата сета да се обединят в един, да се разбъркат и да се извърши ново разделяне с различни представители на двата класа.

// с новите train/validation/test сетове

В този опит се използва техниката transfer learning. (Обяснения за TL и finetuning)

**Опит 1.**

Архитектурата VGG16, като не са включени оригиналните крайни „top“ слоеве, а на тяхно място са поставени подходящи за текущия разглеждания проблем други такива. Като стойности на теглата са заредени получените при трениране върху imagenet сета за данни, комбиниращ… Код на архитектурата:

base\_mdl = krs.applications.vgg16.VGG16(include\_top=False, input\_shape=input\_shape, weights='imagenet')  
base\_mdl.trainable = False  
inputs = krs.Input(input\_shape)  
inp = data\_augm(inputs)  
inp = krs.applications.vgg16.preprocess\_input(inp)  
val = base\_mdl(inp, training=False)  
val = krs.layers.Flatten()(val)  
val = krs.layers.Dense(2048, activation="relu")(val)  
outputs = krs.layers.Dense(1, activation='sigmoid')(val)  
return krs.Model(inputs, outputs)

За да се намали пренагаждането на модела към тренировъчните данни и съответно той да бъде по- устойчив и да се справя добре с невиждани досега изображения, към тренировъчният pipeline е включен и механизмът за обогатяване на данните(data augmentation). Използван е 1 плътно свързан Dense слой с 2048 неврона и relu активираща функция. Като последен слой е поставен същинският класификатор със сигмоидна активационна функция.

Параметри на теста:

Folder: vgg\_2

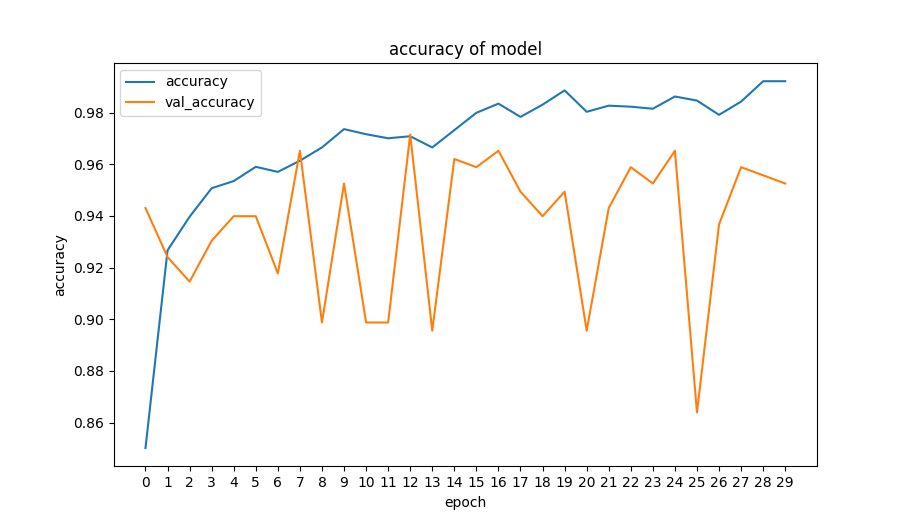
Image size: 128x128x3

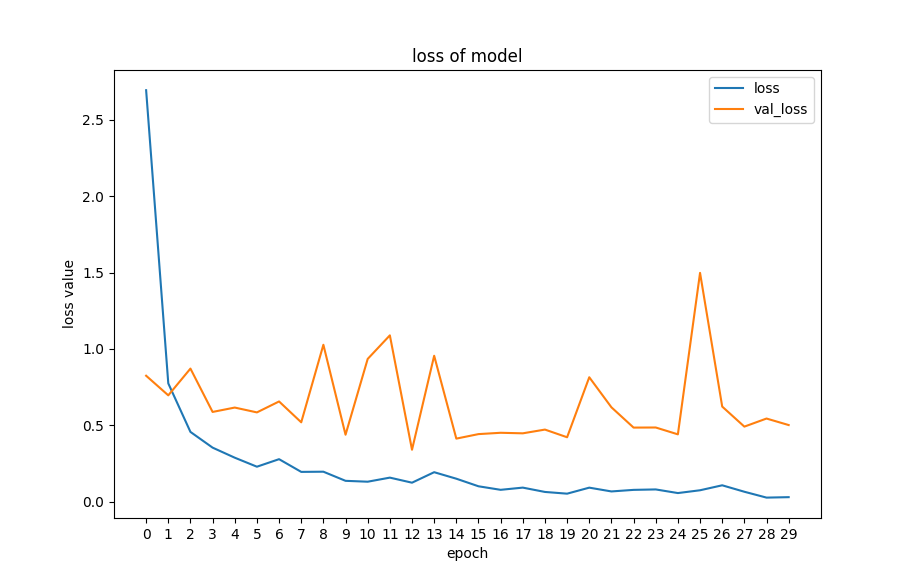
Batch size: 32

Epochs: 30

model.compile(  
 optimizer=keras.optimizers.Adam(learning\_rate=0.001),  
 loss="binary\_crossentropy",  
 metrics=["accuracy", "AUC",  
 keras.metrics.TruePositives(),  
 keras.metrics.FalsePositives(),  
 keras.metrics.TrueNegatives(),  
 keras.metrics.FalseNegatives()])

Резултати от тестването:





От графиките се вижда, че мрежата се учи успешно върху данните за трениране- точността при трениране нараства, а loss стойността- намалява. В стойностите при валидацията, обаче се наблюдава осцилиране на графиката. Например за стойностите на валидационната точност - те варират в интервала 90-96%. В контекста на трансферираното обучение битува и понятието „фина настройка“(fine tuning). Тя включва използване на ниска стойност на параметъра learning rate, защото се очаква предварително натрупаното знание (в този случай заредените тегла от imagenet дейтасета) да спомогне за бързото намиране на оптимално решение. За горния експеримент е зададен learning rate= 0.001- сравнително висока стойност, въпреки използвания адаптиращ оптимизационен метод. Това може да бъде причина за пропускане на оптималното решение(minimum overshooting), резултиращо в осцилирането на графиката.

От резултатите се забелязва, че след епоха 13, валидационната точност започва да спада, докато тази при трениране продължава да се увеличава. Подобна тенденция се забелязва и при стойността на loss функцията. Това е т.нар. пренагаждане на модела спрямо обучителния сет. Въпреки подобряващата се тренираща точност, мрежата започва да се справя по- лошо с невиждани досега данни и губи качеството си да генерализира добре. Съществуват различни техники за справяне с overfitting-a. Една от тях е въвеждането на dropout слой, който изключва част от невроните при обучението(с определена вероятност). На база резултатите и разсъжденията върху тях, следва да бъдат направени следните промени към архитектурата:

* Намаляване на learning rate параметъра на 0.00001
* Добавяне на още един плътносвързан слой
* Добавяне на Dropout слой