**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:

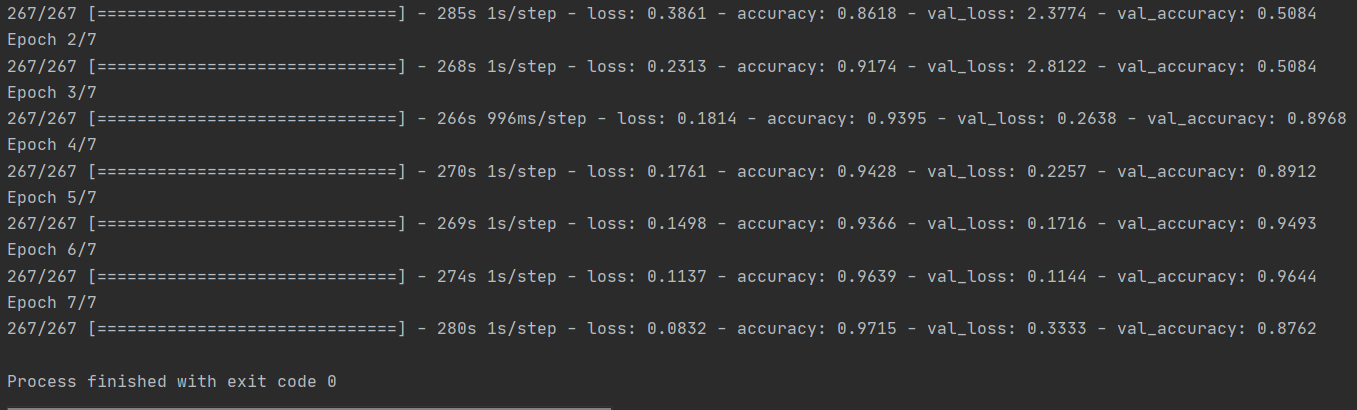
def model\_create\_xception(input\_shape, num\_classes):  
 inputs = krs.Input(shape=input\_shape)  
  
 # Entry block  
 x = krs.layers.experimental.preprocessing.Rescaling(1.0 / 255)(inputs)  
 x = krs.layers.Conv2D(32, 3, strides=2, padding="same")(x)  
 x = krs.layers.BatchNormalization()(x)  
 x = 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:**

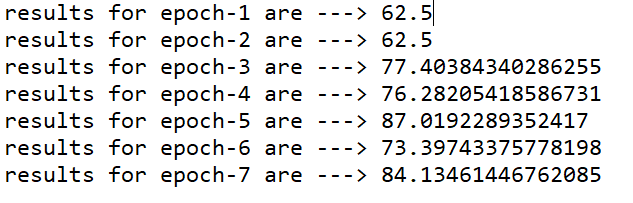
Параметри:

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

Резултати при тренирането:



Резултати при тестването:



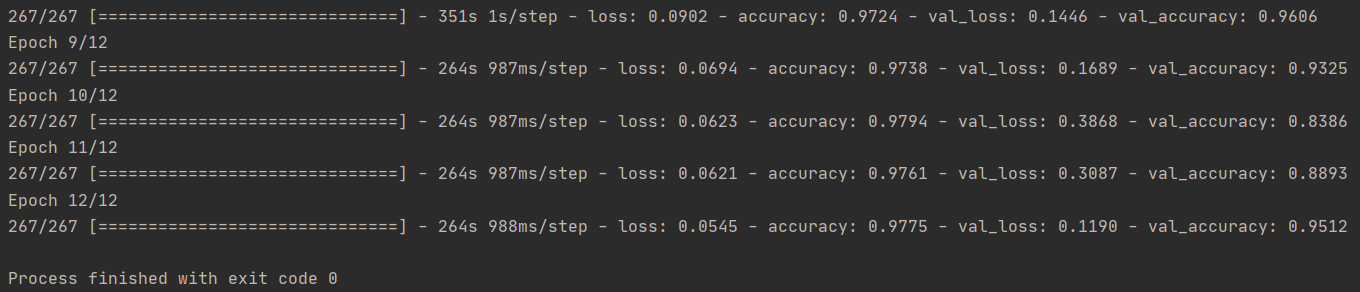
**Run № 2:**

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

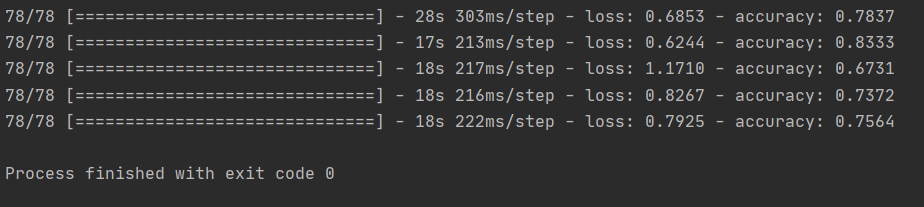
Параметри:

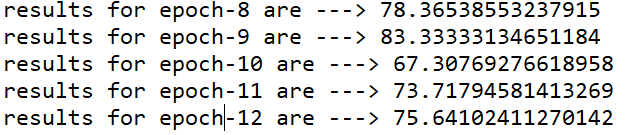
optimizer- Adam(learning\_rate=0.001) ; loss function- binary crossentropy ; **number of epochs – 5(total of 12)**

Резултати при тренирането:

****

Резултати при тестването:



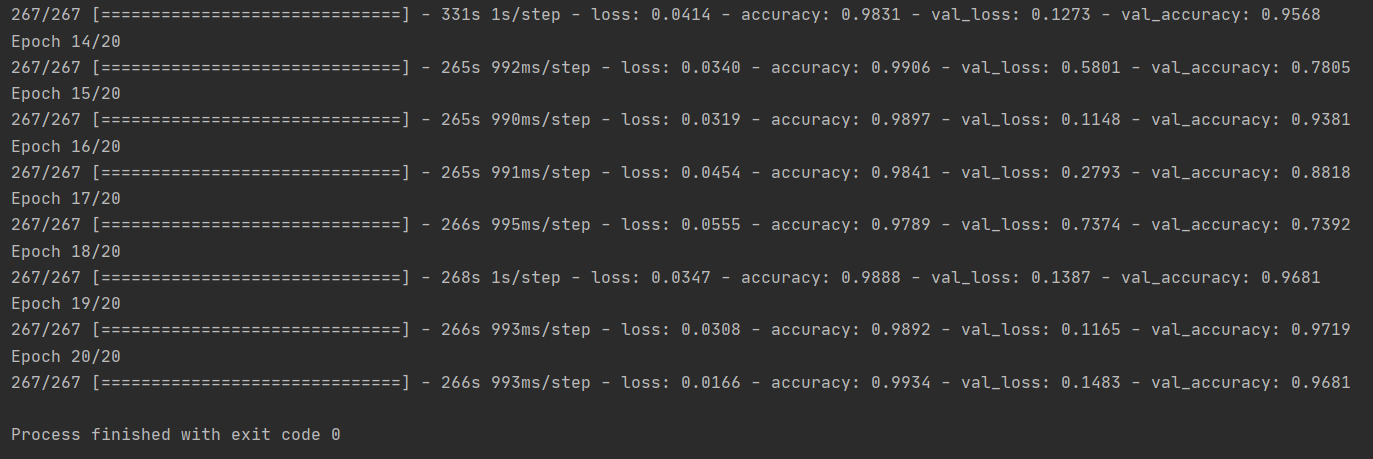


**Run № 3:**

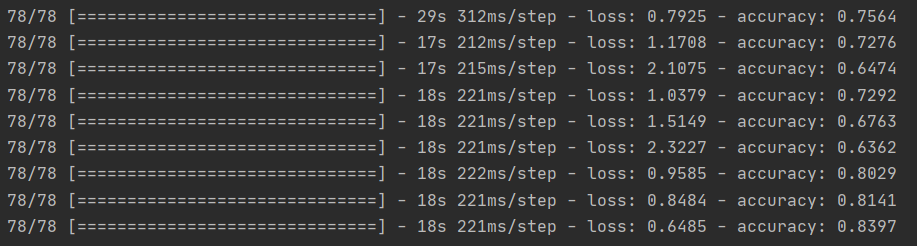
Зареждане на модела от .h5 файл в състояние мрежата- епоха 12 и трениране за още 8 епохи

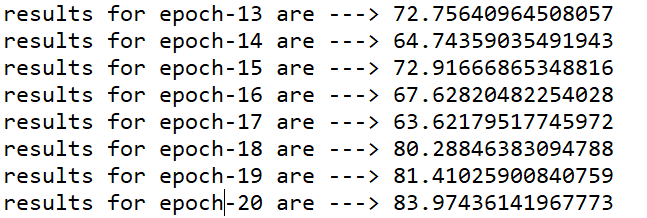
Параметри:

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



Резултати от тестването:



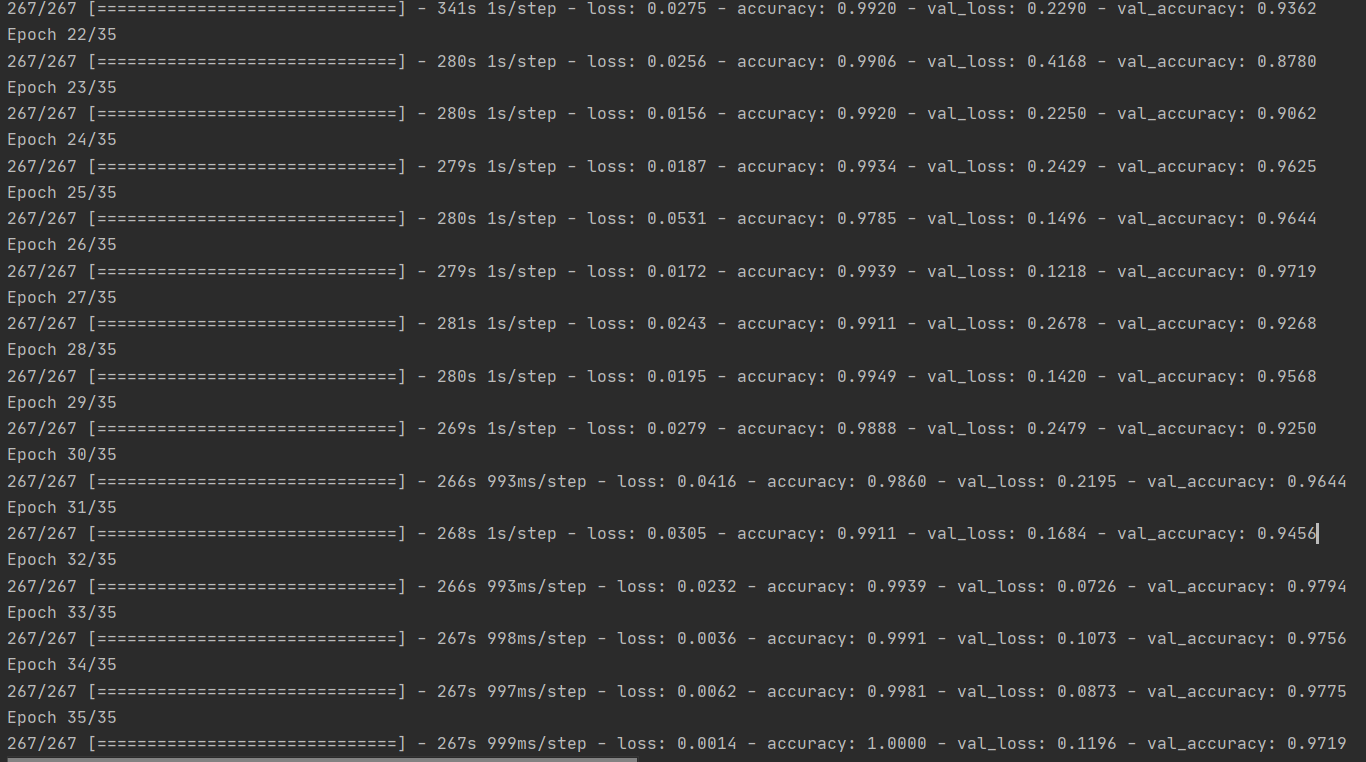


**Run № 4:**

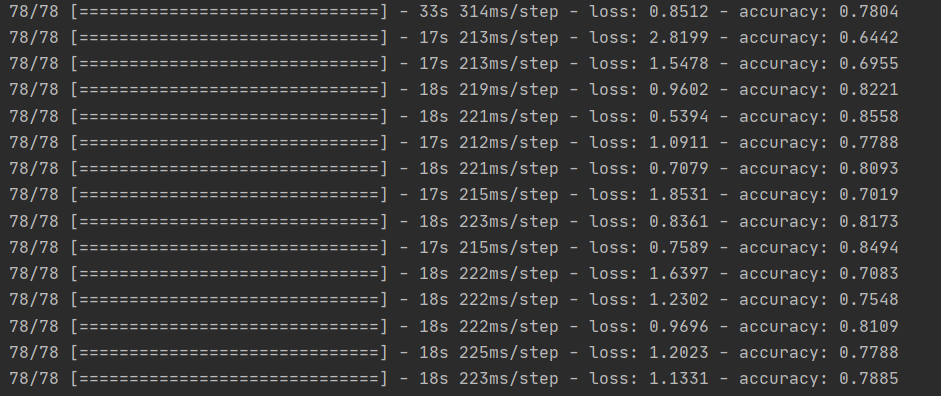
Зареждане на модела от .h5 файл в състояние мрежата- епоха 20 и трениране за още 15 епохи

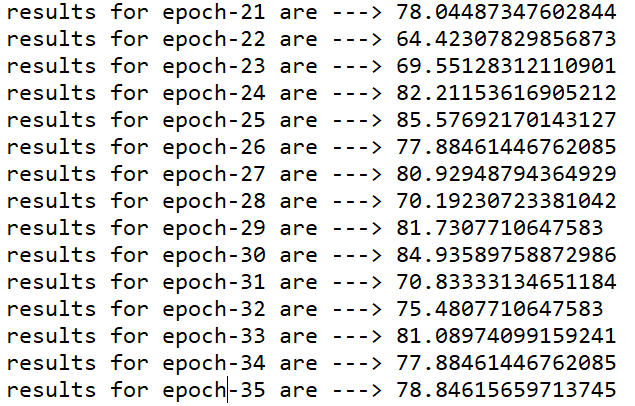
Параметри:

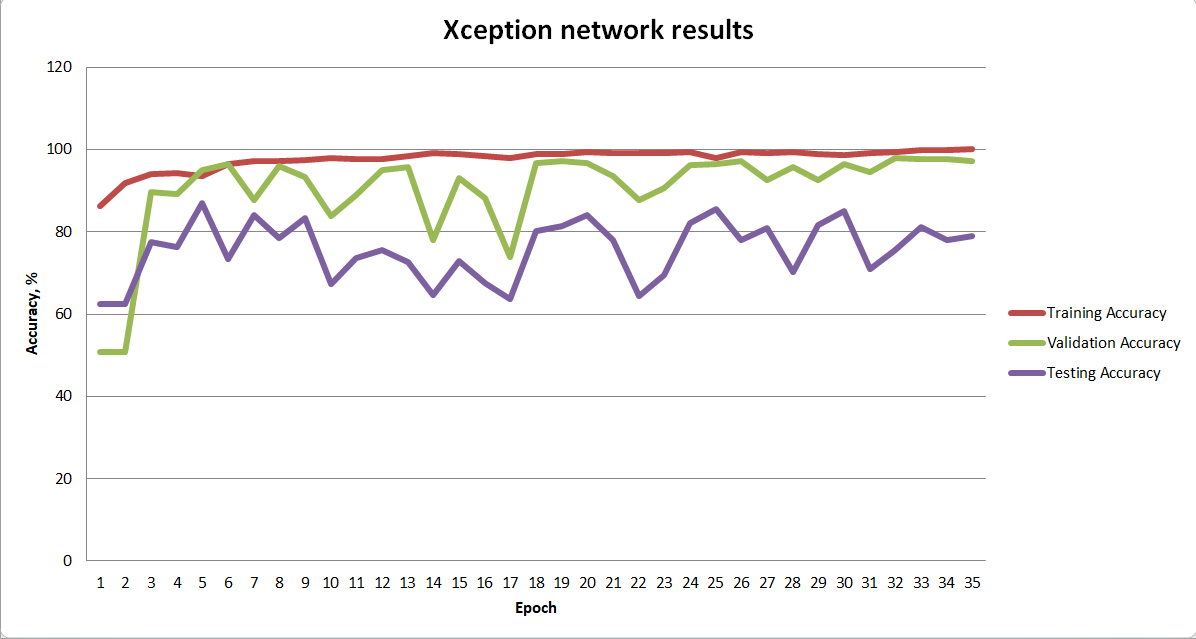
optimizer- Adam(learning\_rate=0.001) ; loss function- binary crossentropy ; **number of epochs – 15(total of 35)**

****

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

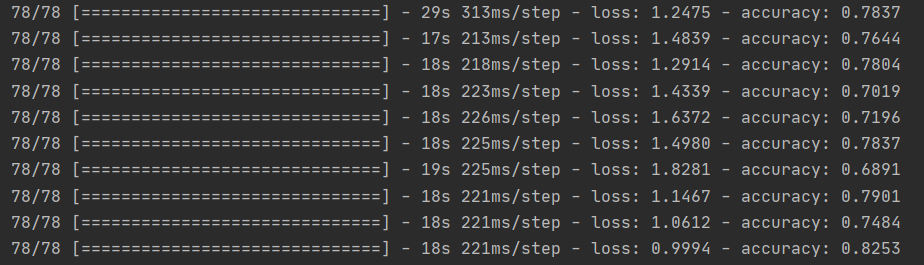
Параметри:

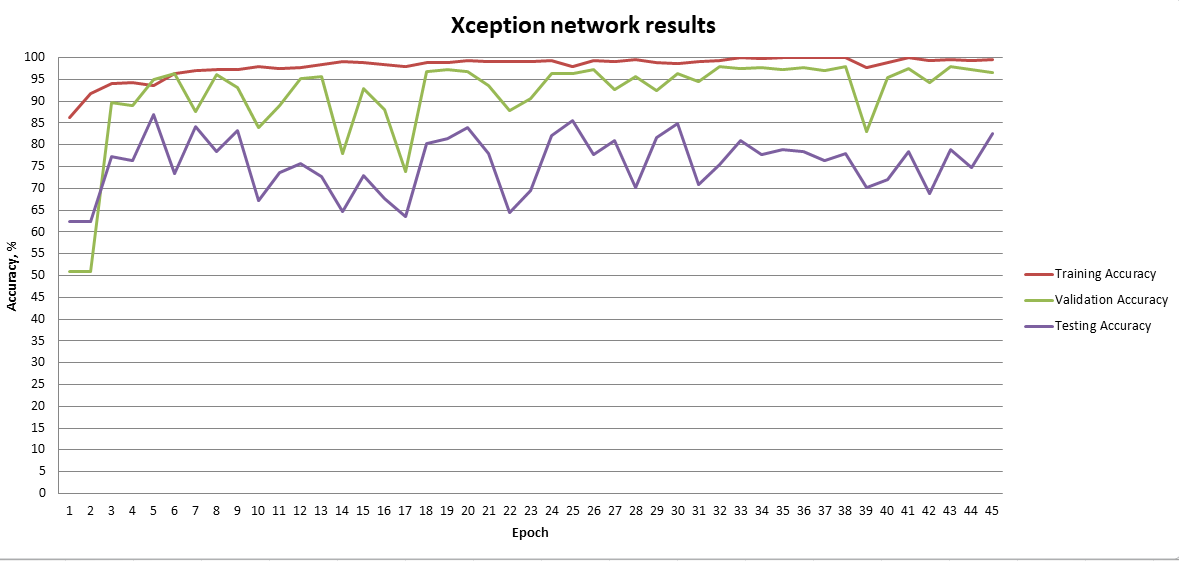
optimizer- Adam(learning\_rate=0.001) ; loss function- binary crossentropy ; **number of epochs – 10(total of 45)**

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

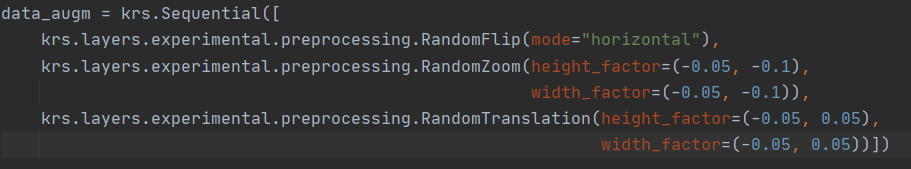
3. Промяна в архитектурата на мрежата.

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

**Опит 2:**

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

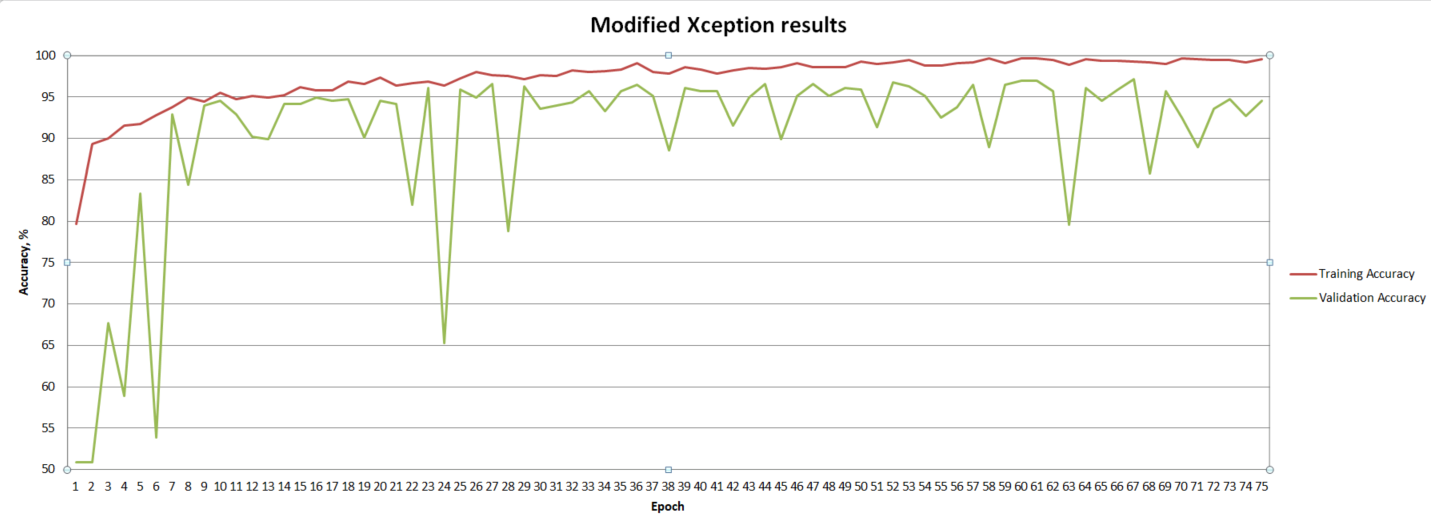
* Добавяне на data augmentation **към архитектурата на невронната мрежа(**обогатяването се извършва „on-line“ върху GPU като слоеве от мрежата**)**

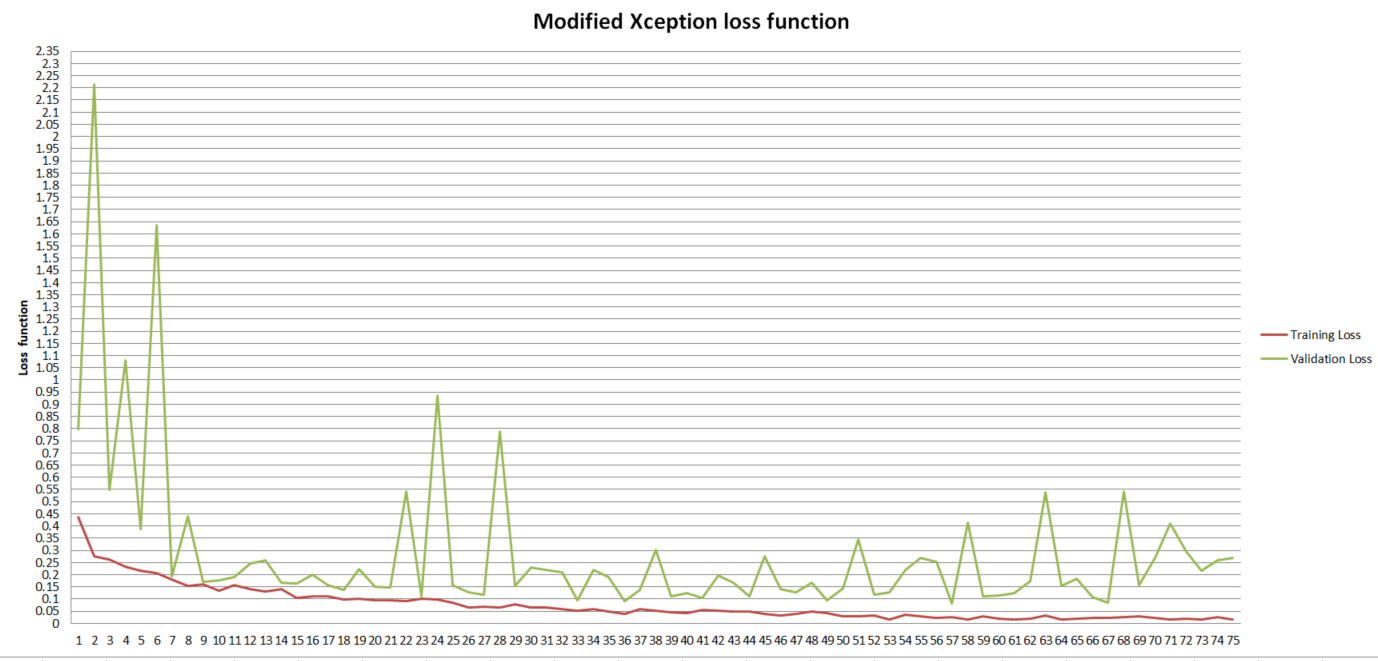


* Промяна на стойността на вероятността на DropOut слоя в мрежата на 0.6 (предишна стойност – 0.5)
* Промяна на хиперпараметъра learning rate на оптимизационния метод Adam на 0.00005(предишна стойност 0.001)
* Стойността на batch size е запазена същата- batch size= 8
* Зададено е отпечатване на следните 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.

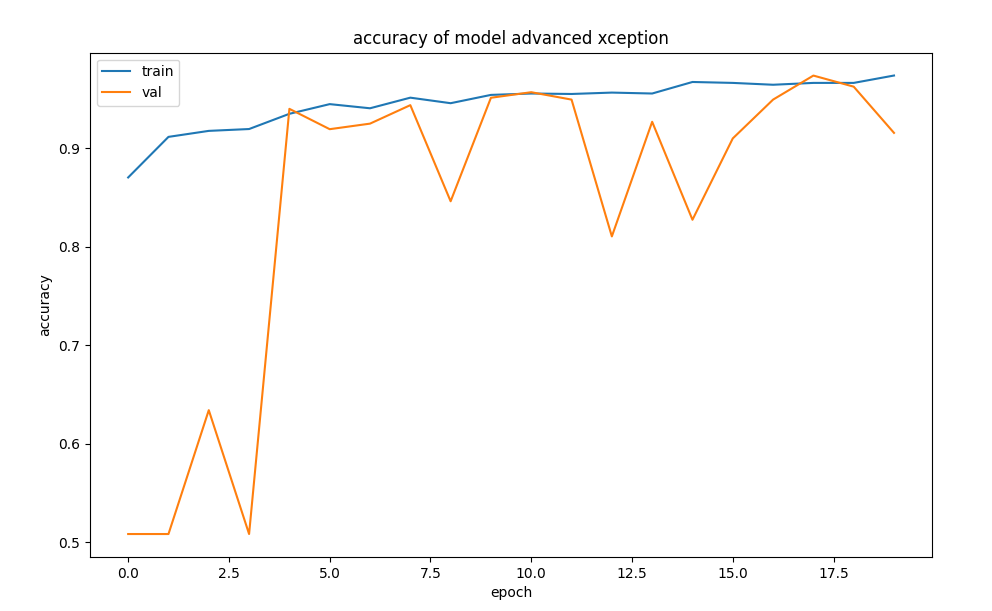
**Run № 2:**

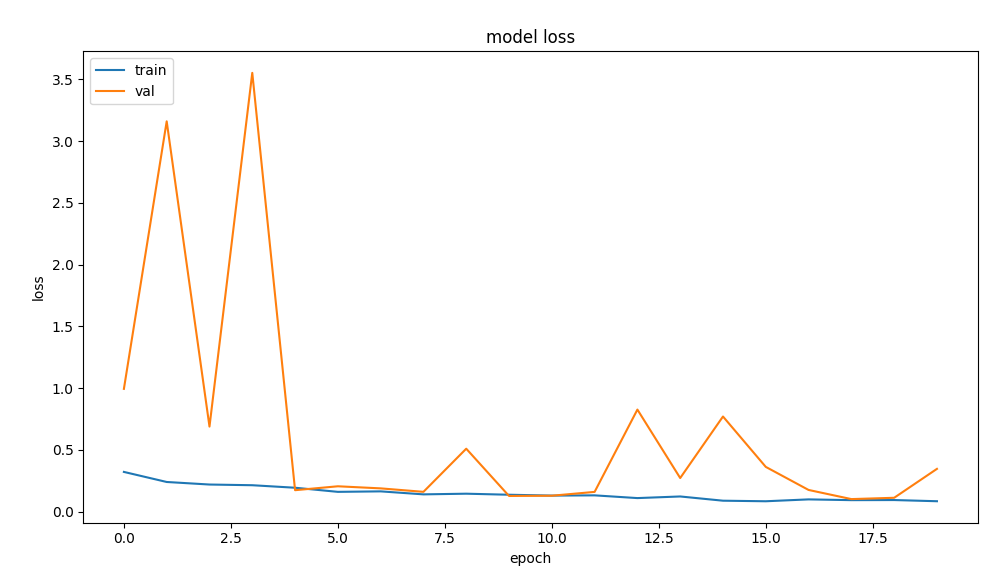
Използвана е същата архитектура на НМ, но с друг оптимизиращ алгоритъм- RMSProp:

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.

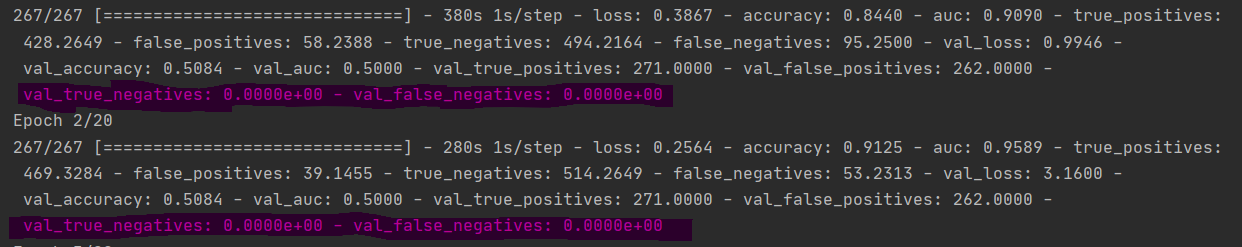
Резултати:



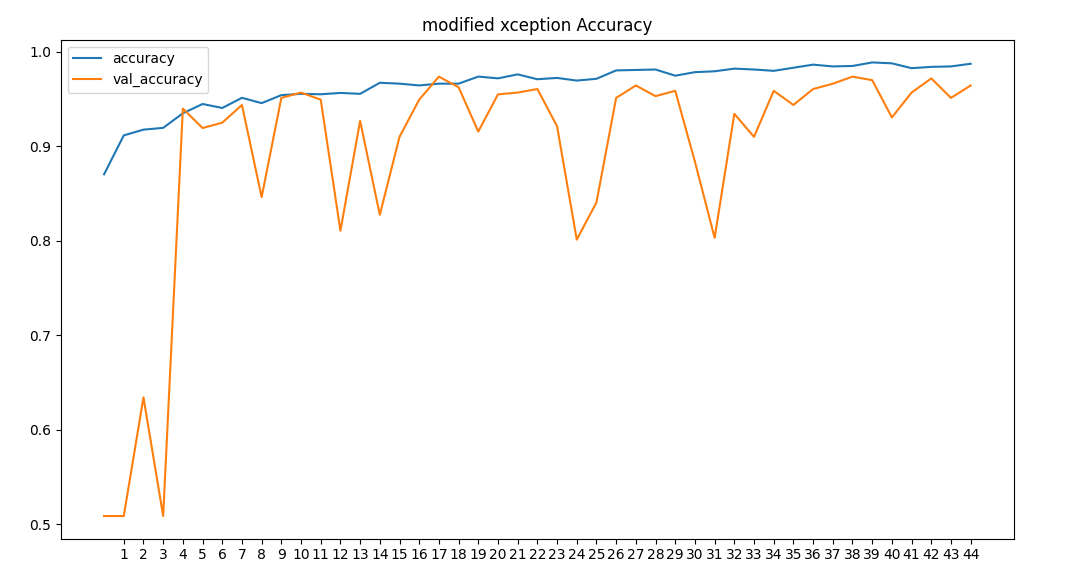


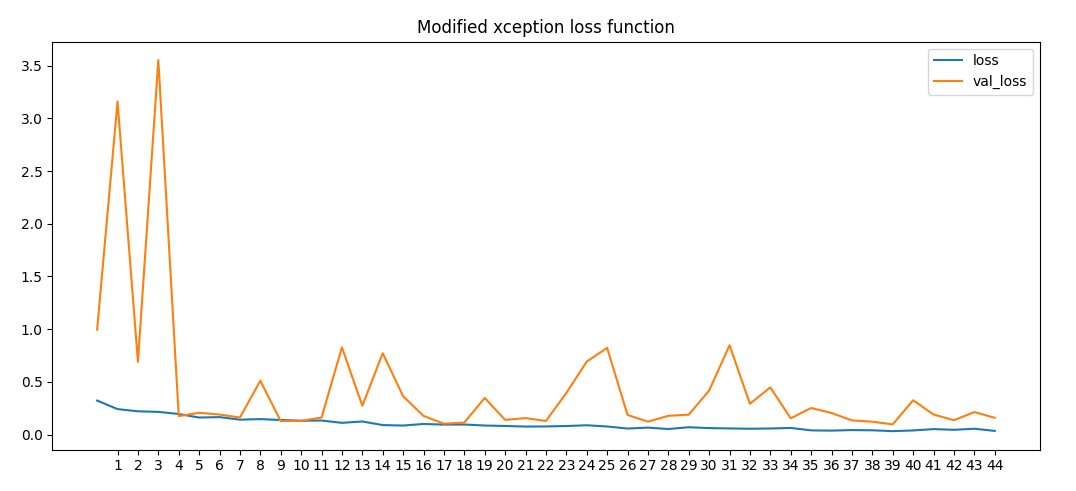
Осцилирането отново е по- малко спрямо Опит 1, но са нужни още данни(още трениране на мрежата), за да се направи заключение и да се открие ясна тенденция.

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



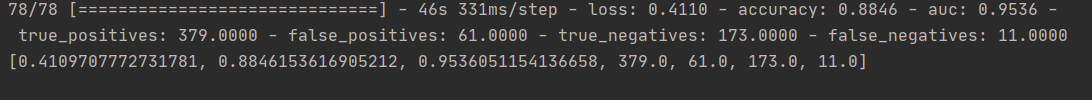
Следва трениране на мрежата за още 25 епохи(общо 45)





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:

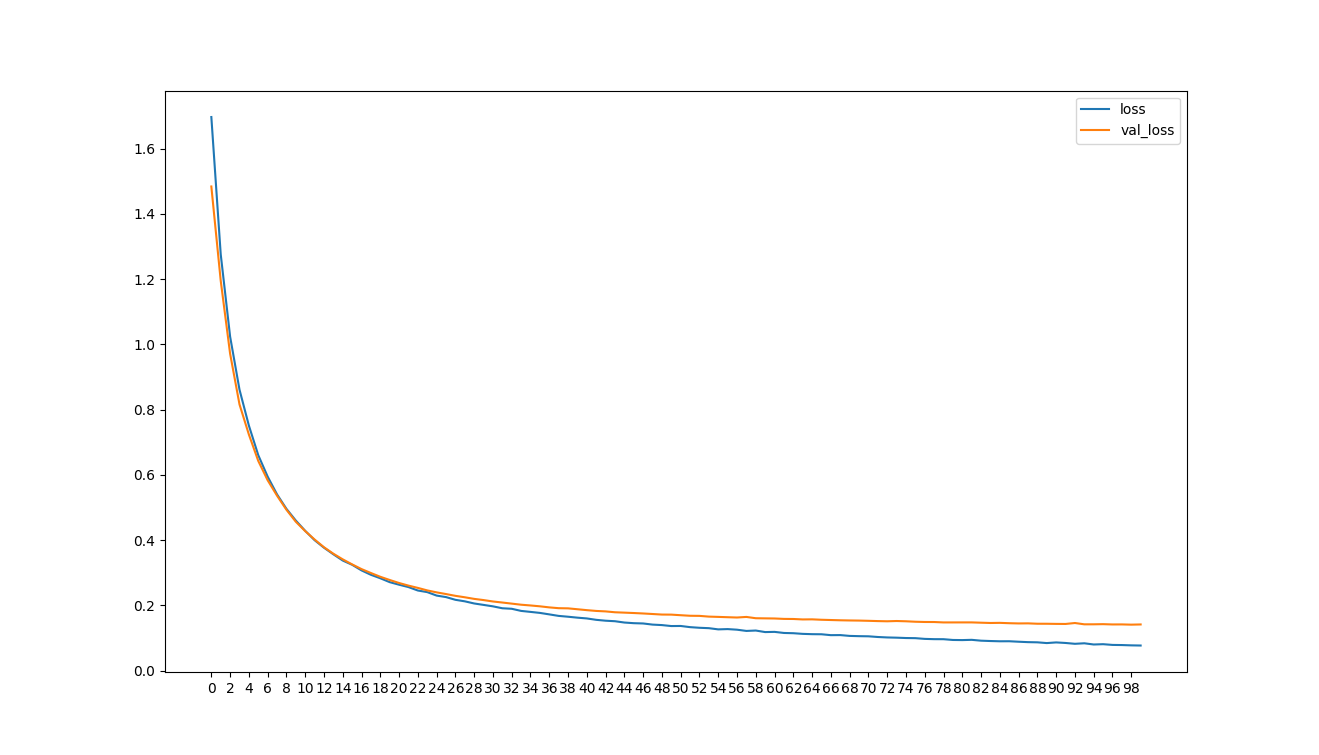
* GlobalAvaragePooling2D- пулинг слой
* Dense(1,activation=’sigmoid’)- изходен слой отразяващ резултата от мрежата в рамките на 1 неврон, чрез който се извършва класификацията. Като активираща функция е зададена сигмоидната, удобно ограничаваща резултата в интервала [0,1].

Извършено е трениране за 100 епохи, със следните параметри :

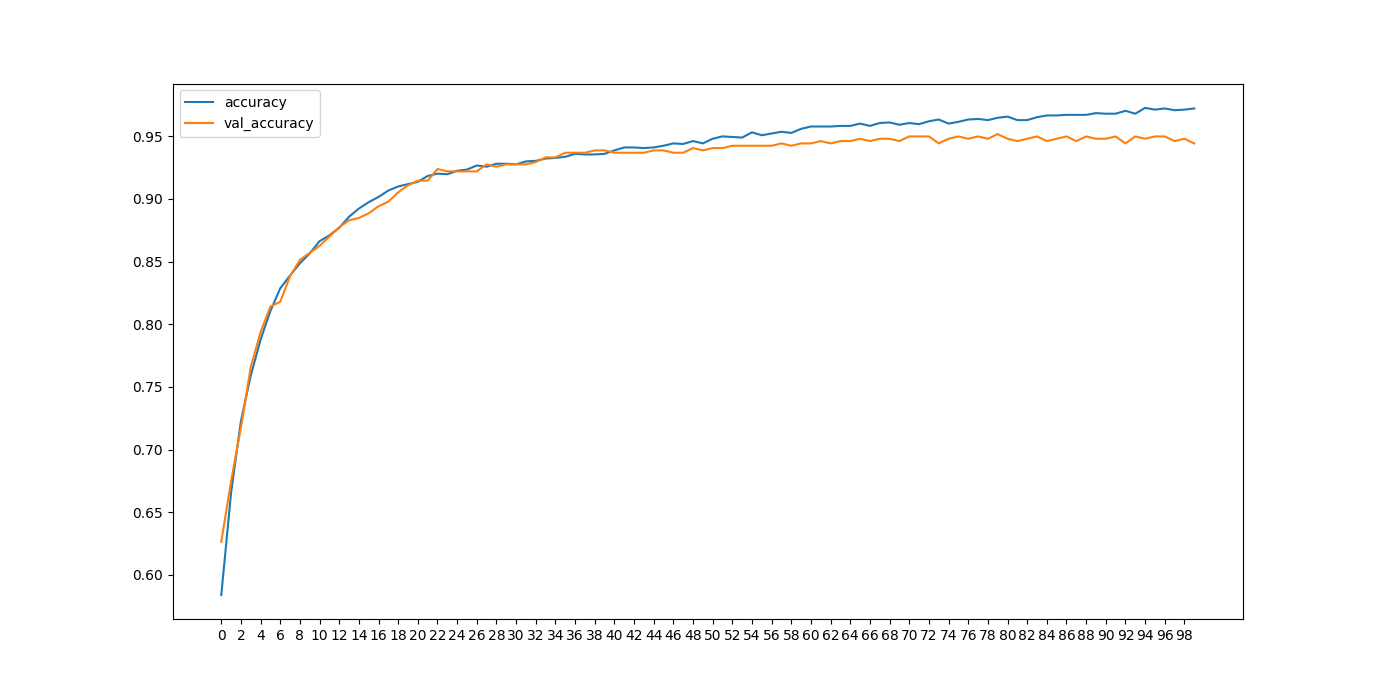
optimizer=keras.optimizers.Adam(learning\_rate=0.0001),  
loss="binary\_crossentropy"

При което се получават следните резултати:

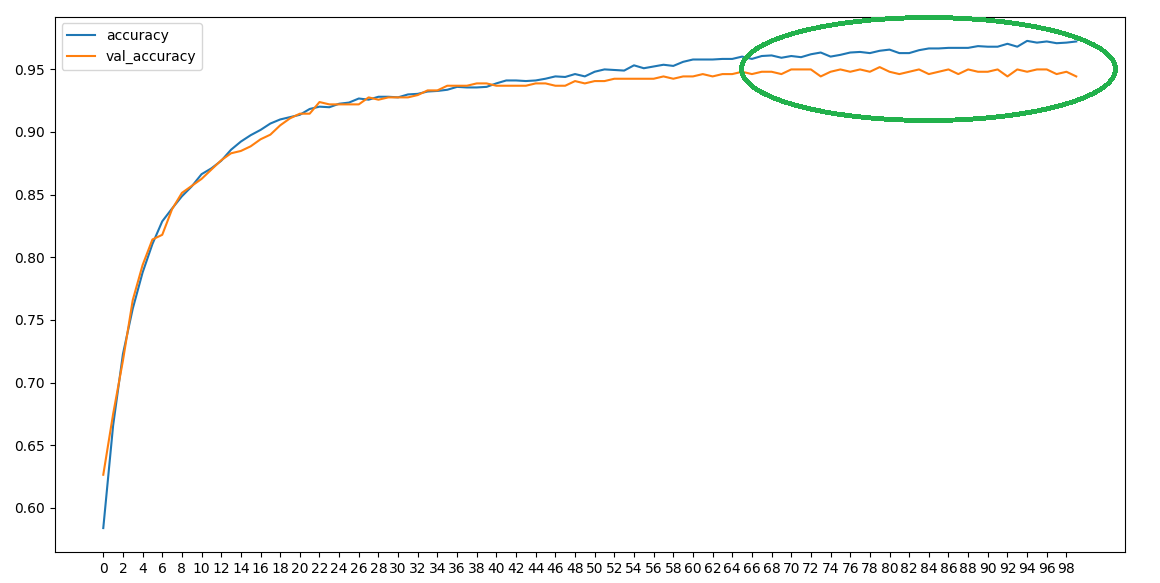
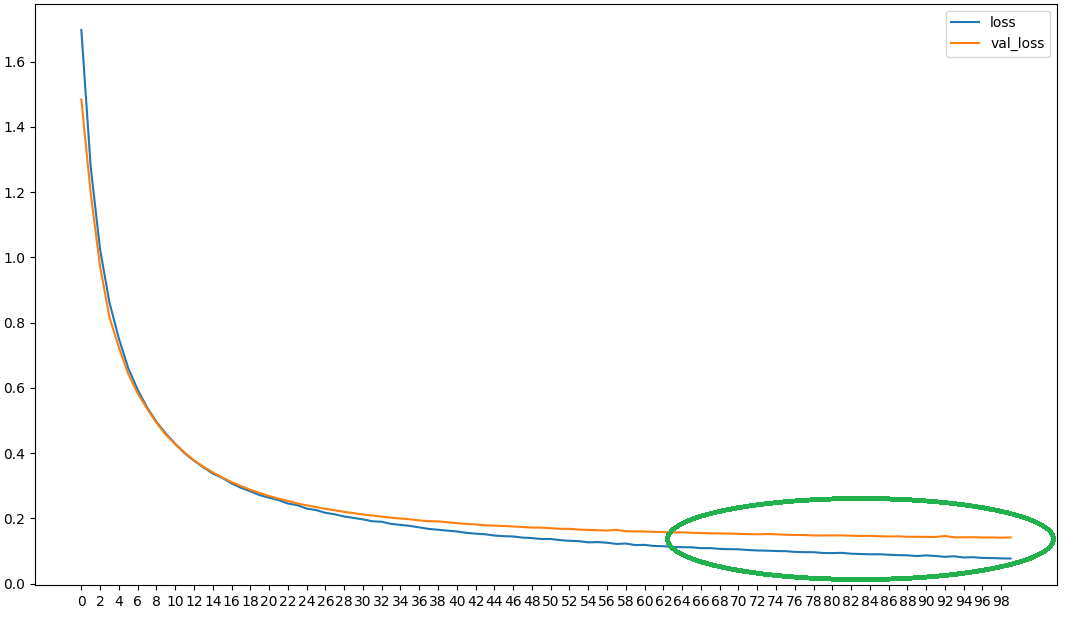
* За ценовата функция



* 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.**

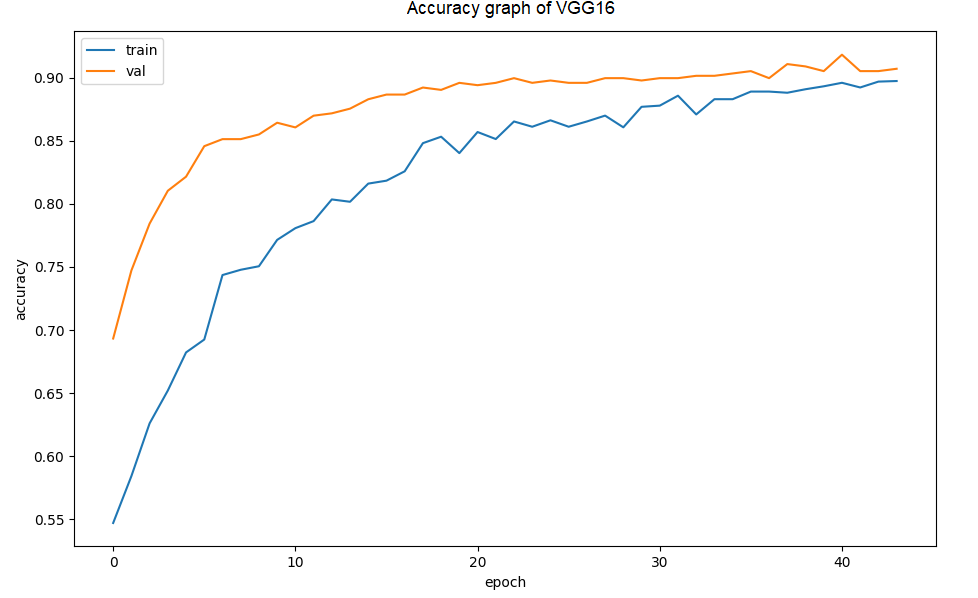
Код на създаване на модела:

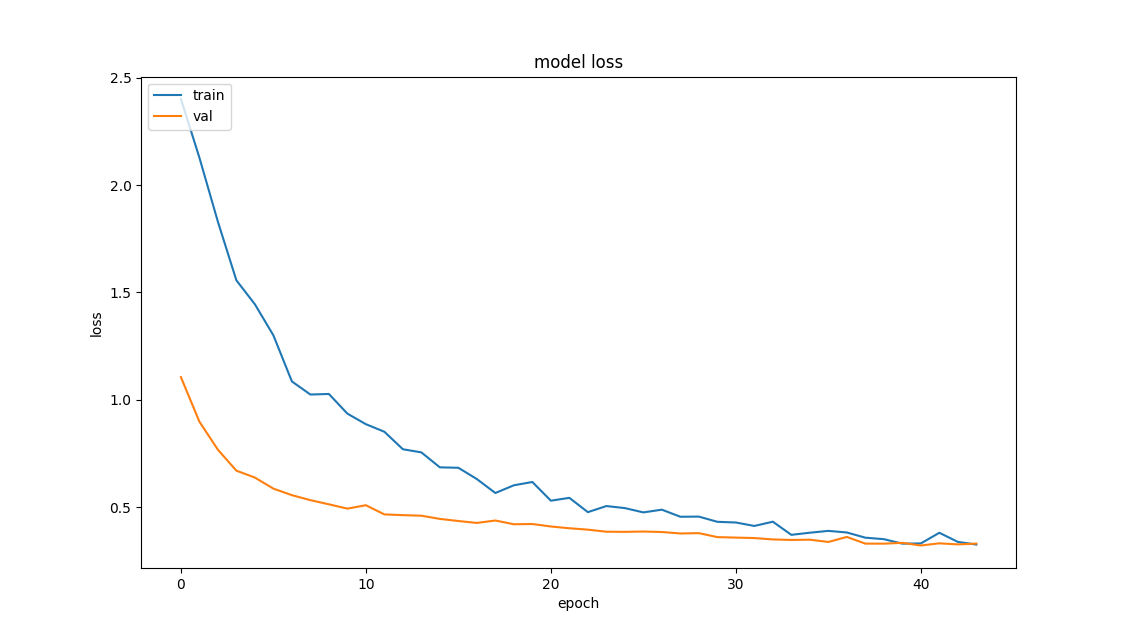
def model\_create\_vgg16(input\_shape):  
 base\_mdl = krs.applications.vgg16.VGG16(include\_top=False, Input\_shape=input\_shape, weights='imagenet')  
 base\_mdl.trainable = False  
 inputs = krs.Input(input\_shape)  
 inp = data\_augm(inputs)  
 inp = krs.applications.vgg16.preprocess\_input(inp)  
 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 :

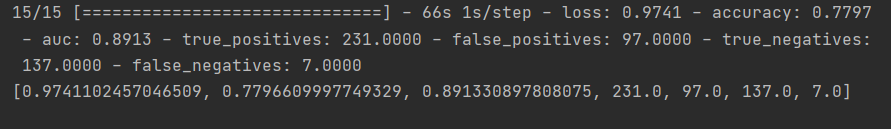




От представените графики на резултатите за стойността на ценовата функция и точността се вижда, че техниката за ранно спиране на тренирането е прекратила процеса в епоха 43(не е имало подобрение в епохите 41,42,43). Наблюдава се и минимално осцилиране, което най-вероятно се дължи на добавеното обогатяване 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 :



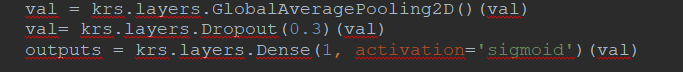
Достигнатата точност при тестовия дейтасет е 77.97%. Въпреки високото ниво сгрешени предположения от мрежата, по-тежкият вид грешка(Type 2 error- фалшиво отрицателни стойности) е доста по-малка спрямо Type 1 error- фалшиво положителните:

* 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:



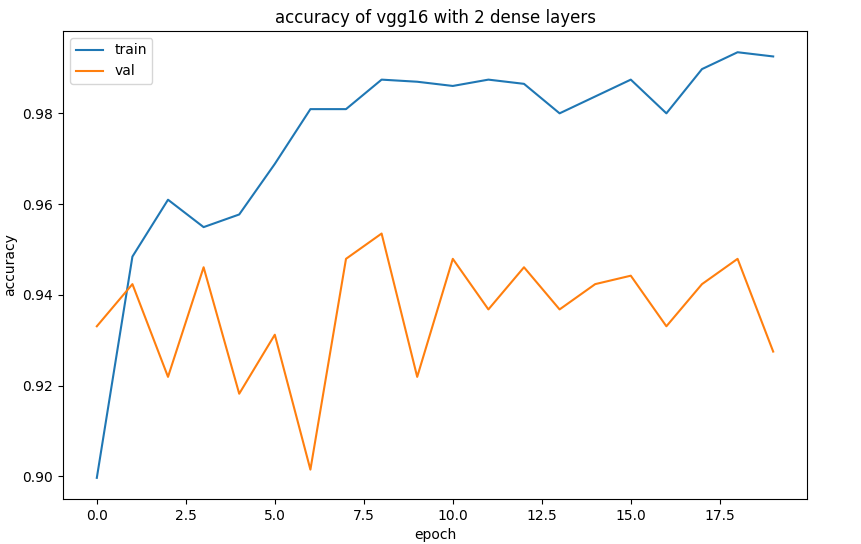
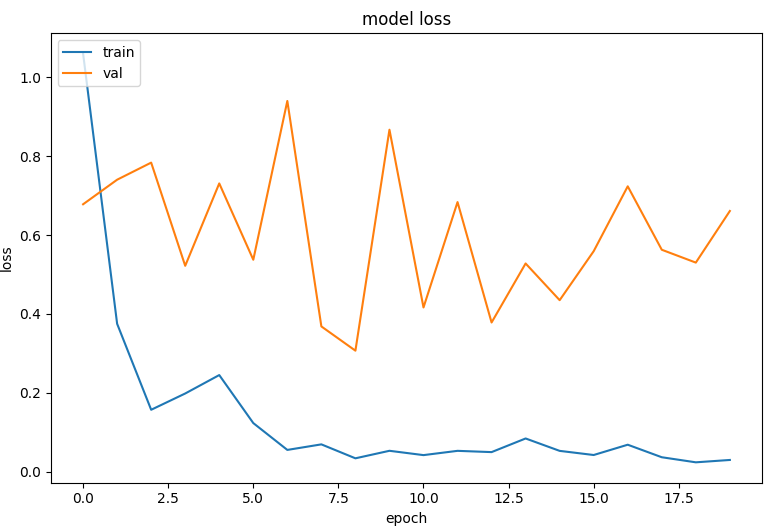
Както се вижда от графиките на предното пускане, точността, която се достига при трениране/валидация е от порядъка на 80-90%, а тази при тестване с тестовия дейтасет- дори по-ниска. Много вероятно е изходните слоеве да не успяват да усвоят достатъчно добре специфичността на данните. Затова ще бъде направена 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.

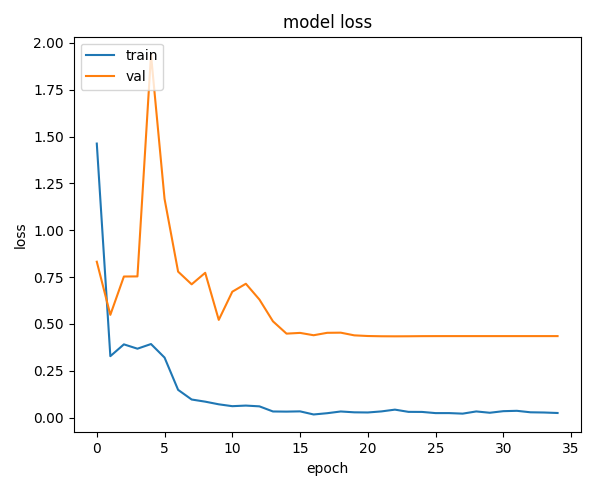
**Dropout**

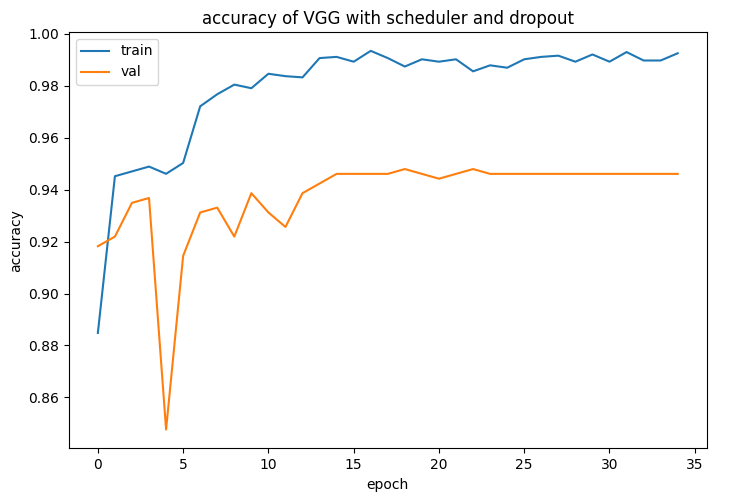
Добавяне на dropout слой като мярка за регуляризация и респективно намаляване на пренагаждането на модела:

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)

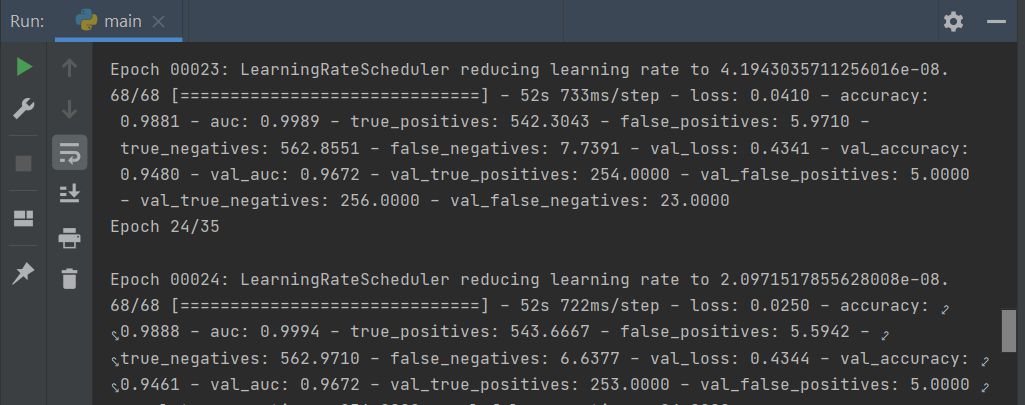
Поставен е dropout слой с вероятност 0.3 между двата плътносвързани слоя в края на мрежата. Като параметър е подаден и seed, за да може да бъде пресъздадено същото действие на слоя в следващи опити.

**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

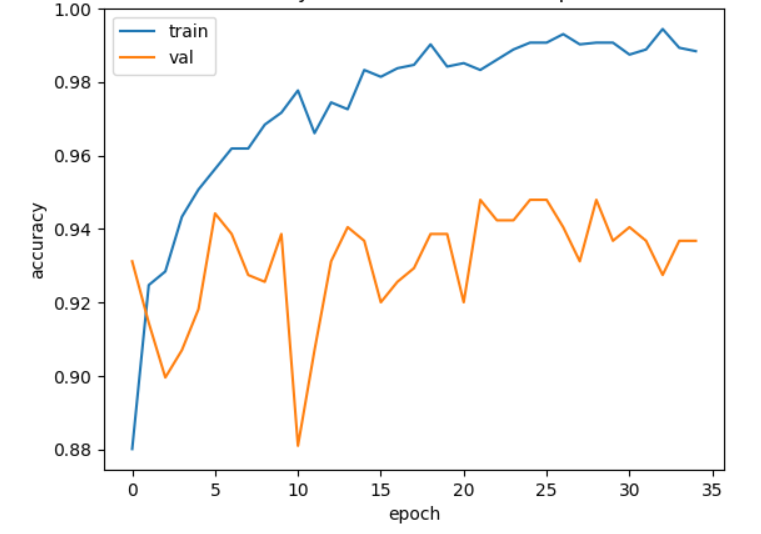
MIN\_LEARNING\_VALUE = 0.0000001

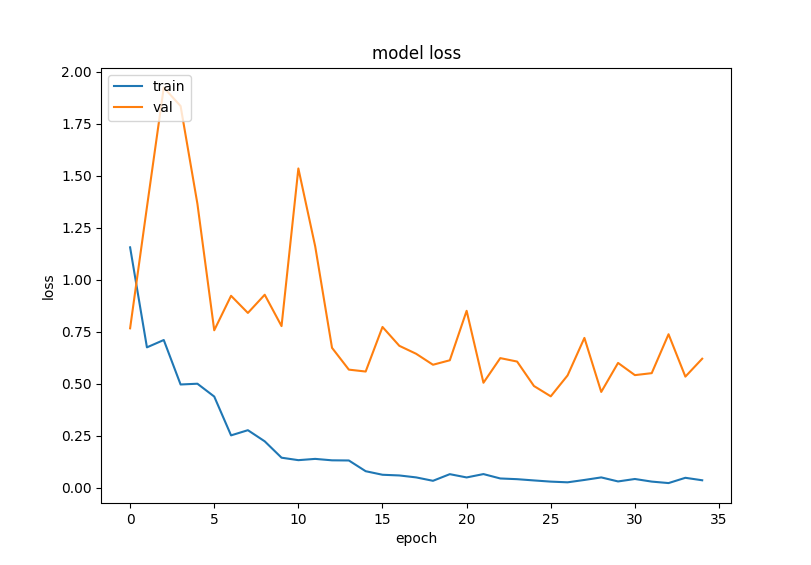
def lr\_scheduler(epoch, lr):  
 if epoch < 5 or lr < MIN\_LEARNING\_VALUE:  
 return lr  
 elif 5 <= epoch < 15:  
 return lr\*0.9  
 else:  
 return lr\*0.97

1. Увеличаване на dropout вероятността от 0.3 на 0.5

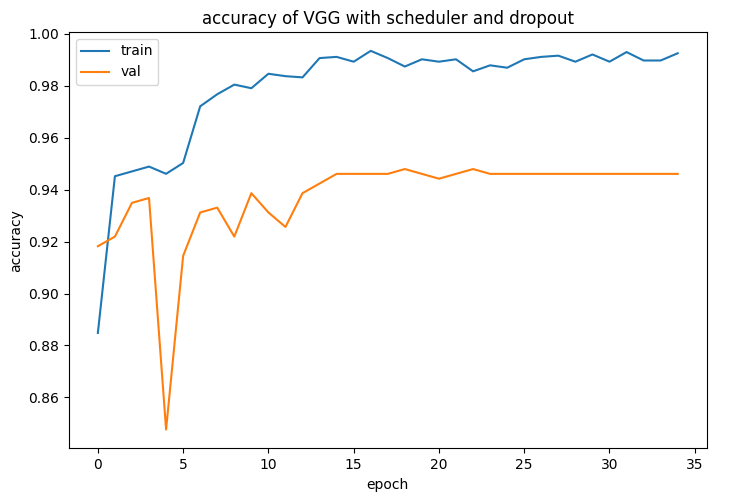
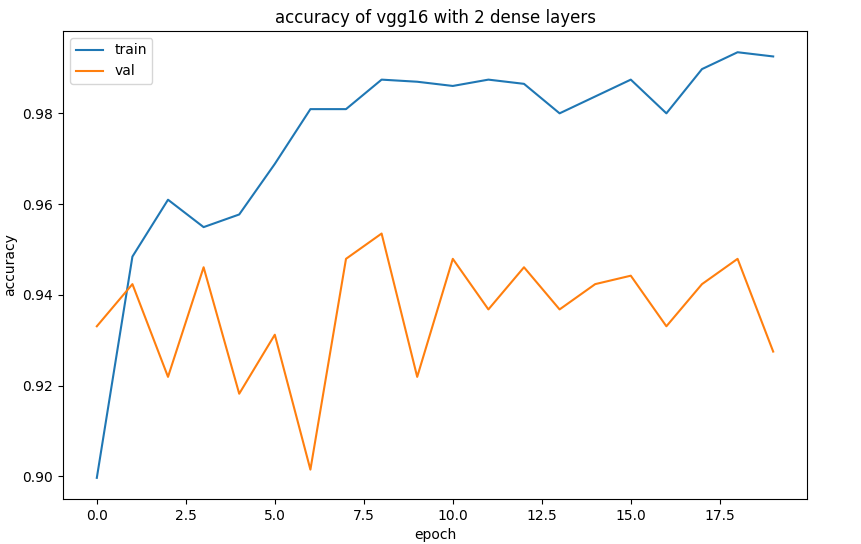
al = krs.layers.Flatten()(val)  
val = krs.layers.Dense(2048, activation="relu")(val)  
val = krs.layers.Dropout(0.5, seed=1333)(val)  
val = krs.layers.Dense(2048, activation="relu")(val)

**Резултати:**

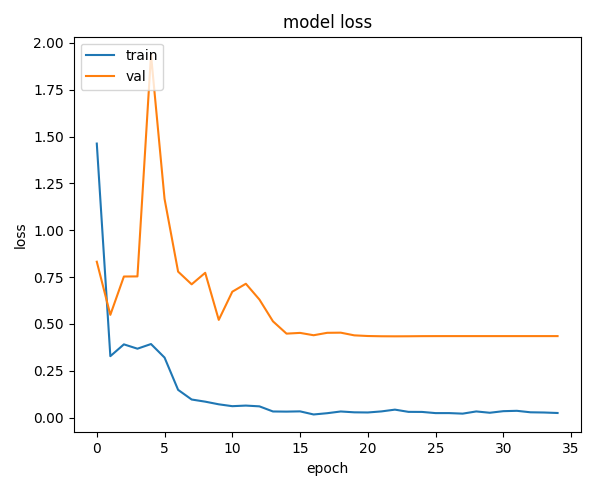
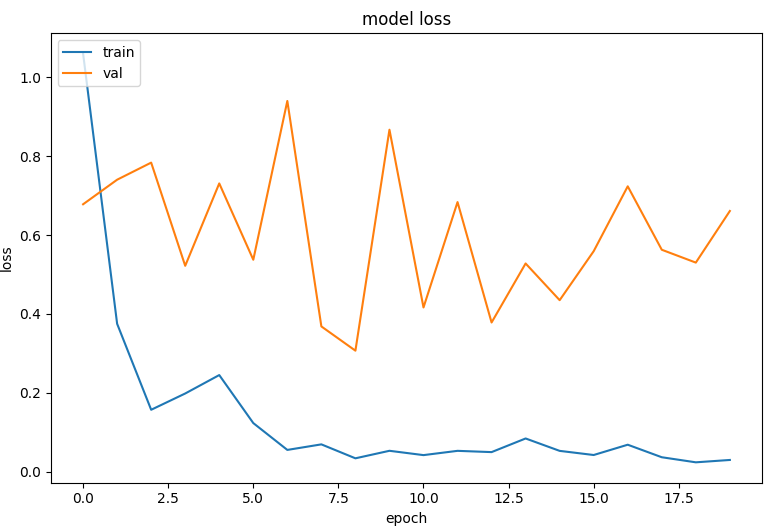
****

****

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

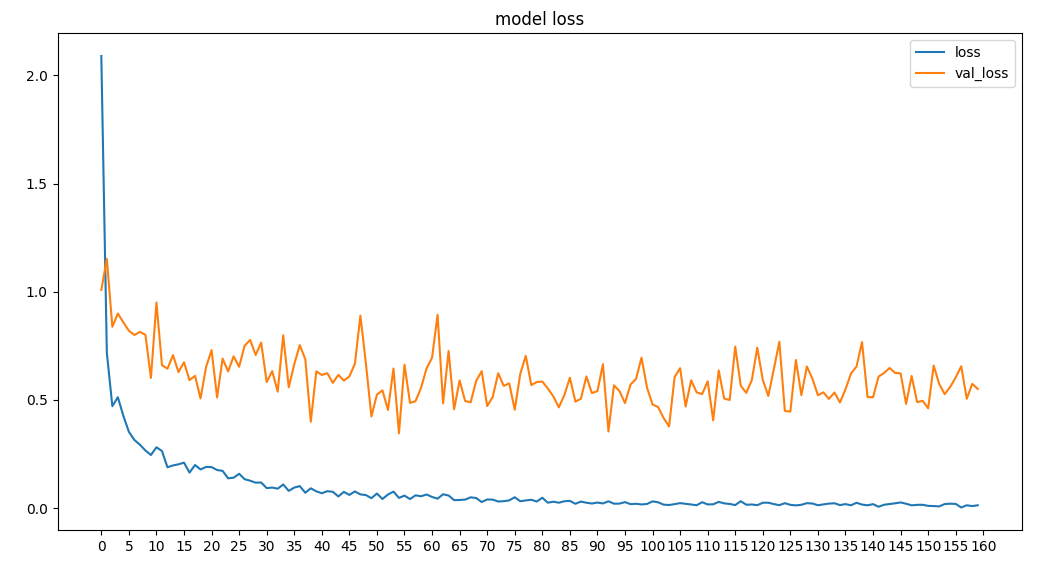
**Run №1:**

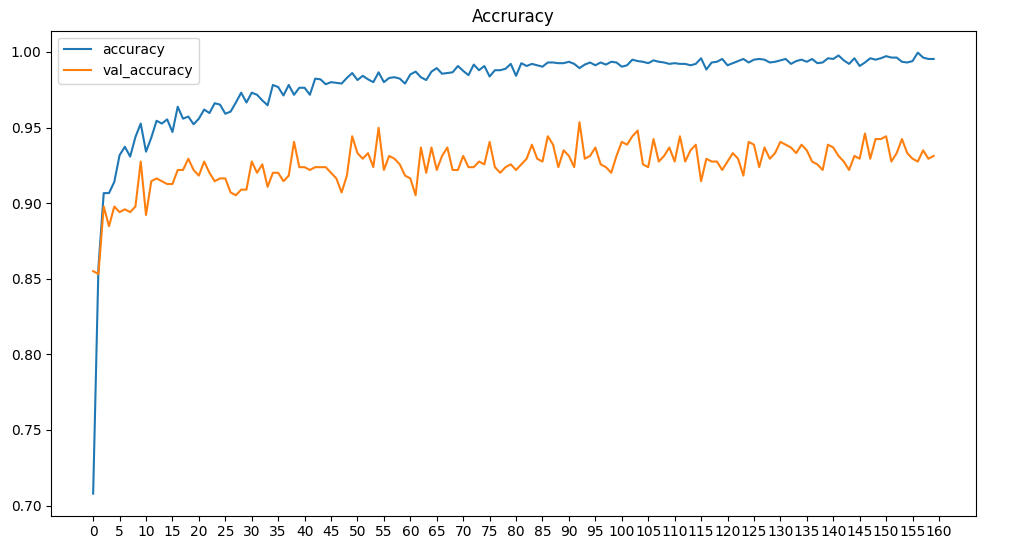
Параметри на текущото трениране:

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:**

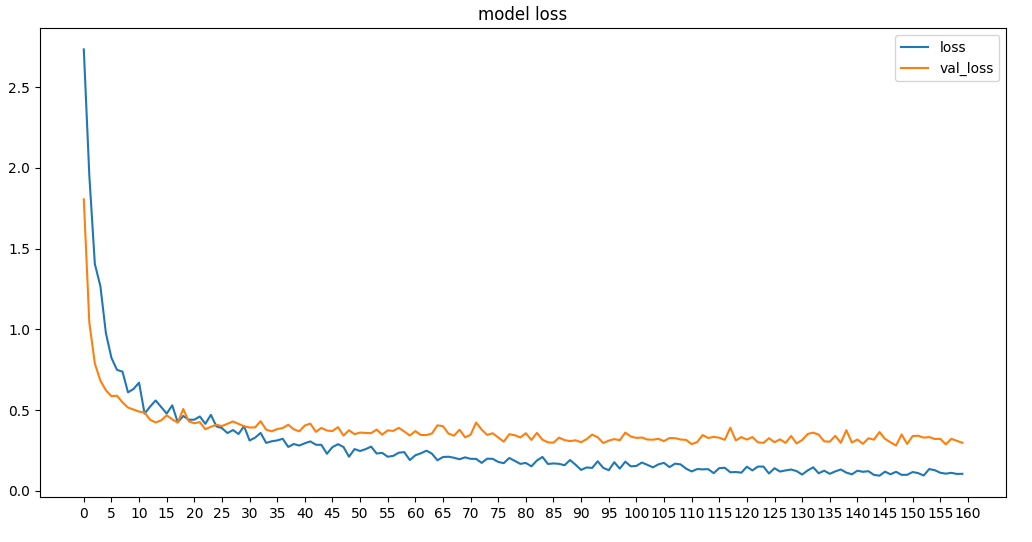
Параметри на текущото трениране:

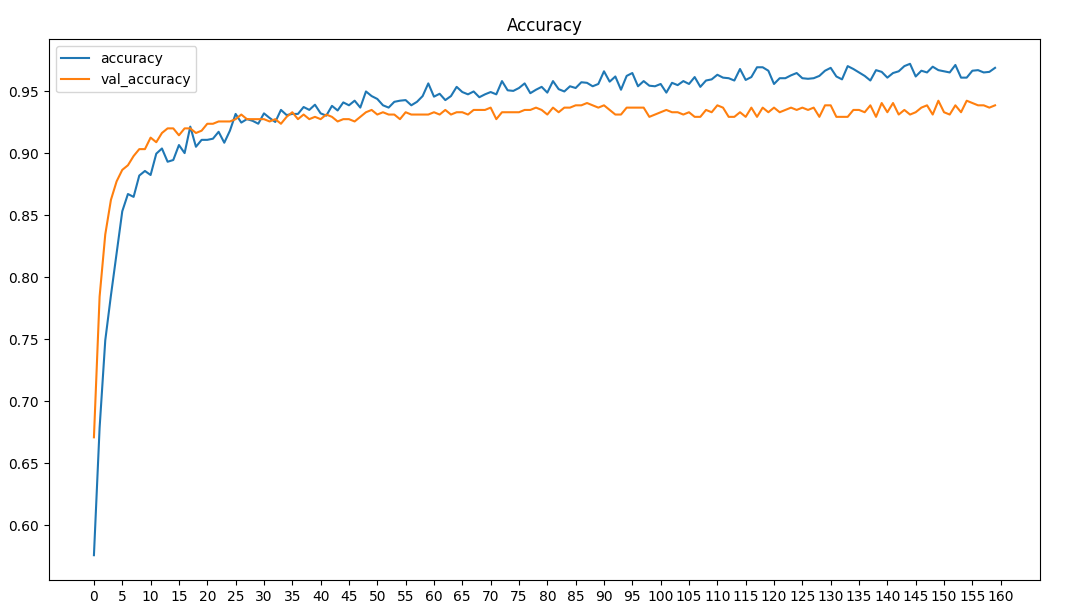
Optimizer: **Adam**

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).

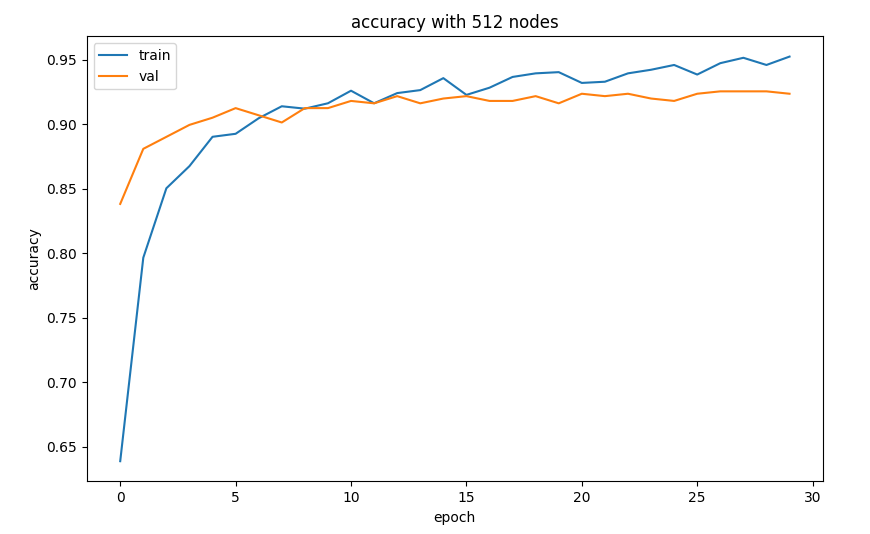
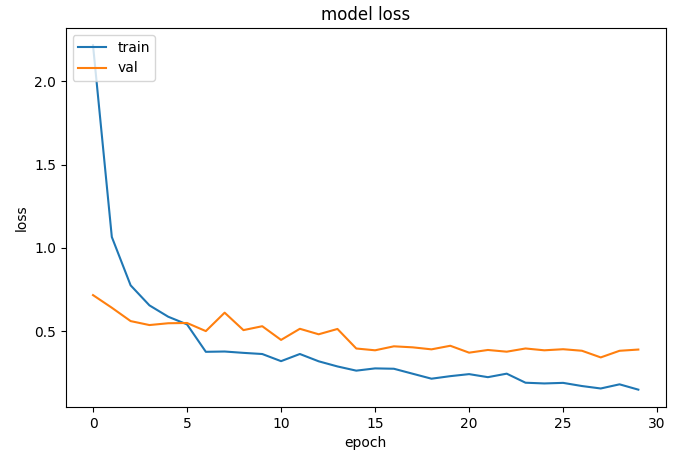
val = krs.layers.Dense(512, activation="relu")(val)

Параметри на тестването:

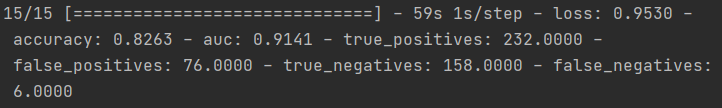
Брой епохи: 30

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.

**Тест 1.**

Параметри на теста:

Folder: vgg\_2

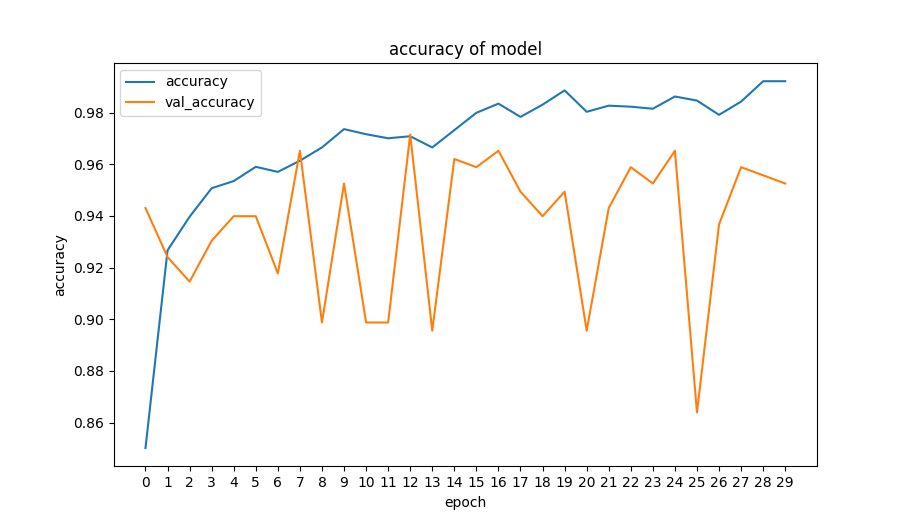
Image size: 128x128x3

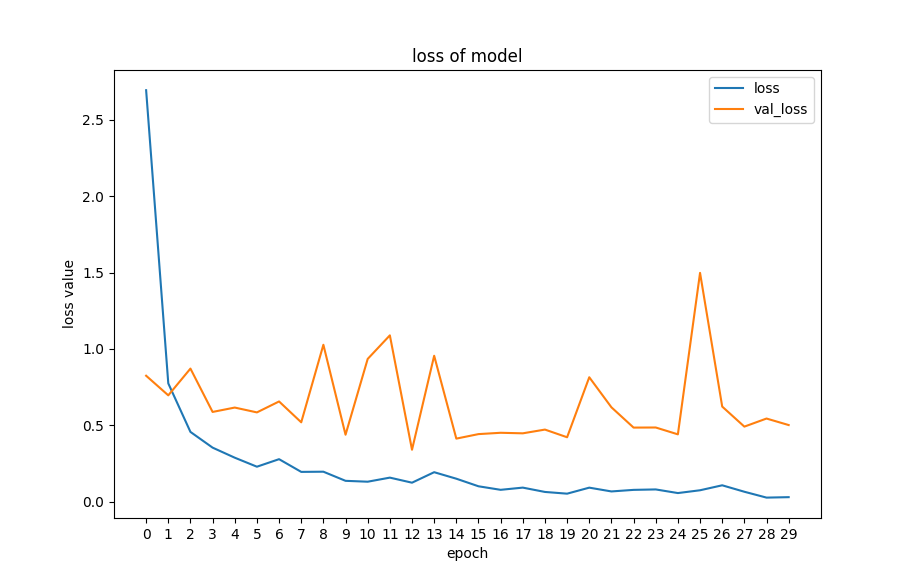
Batch size: 32

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(optimum 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/обучителния процес:

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

**Тест 2.**

В този тест са отразени промените в архитектурата, изяснени по-горе:

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)  
val = krs.layers.Dropout(0.1, seed=1333)(val)  
val = krs.layers.Dense(2048, activation="relu")(val)  
outputs = krs.layers.Dense(1, activation='sigmoid')(val)  
return krs.Model(inputs, outputs)

Добавен е втори Dense слой с 2048 неврона и ReLu активационна функция. Между двата плътносвързани слоя е добавен Dropout слой с вероятност за включване- 0.1.

Освен промените в архитектурата, използвана е и по-ниска стойност на параметъра learning rate= 0.00001. Така параметрите за теста са следните:

Folder: vgg\_4

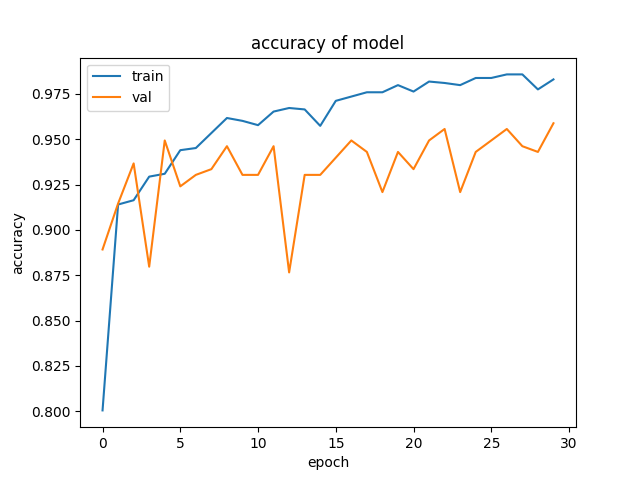
Image size: 128x128x3

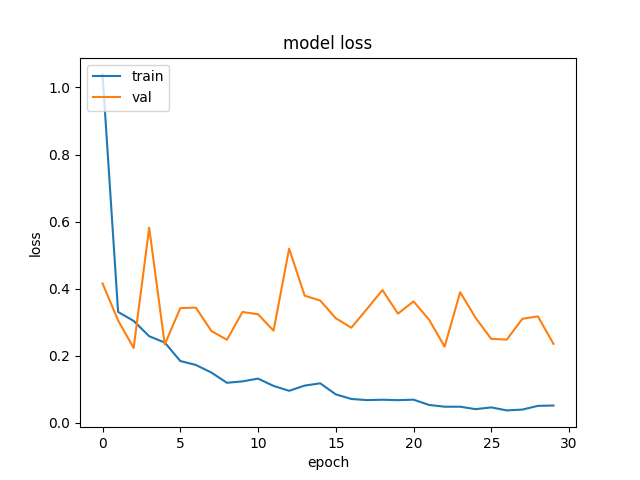
Batch size: 32

Epochs: 30

Learning rate: 0.00001 (1.10-5)

Резултати:





Забелязва се известно подобрение спрямо предните резултати. Пренагаждането на модела е намалено- в графиката за точността се наблюдава лека тенденция за повишаване по време на валидация, но все пак след епоха 8 се запазва разлика между валидационна/тренираща точност от порядъка на 2-3%. В графиките на ценовата функция, обаче не личи значим прогрес след епоха 5 за валидационната точност, докато мрежата успешно се обучава и достига стойности от 0.06 при трениране. Необходим е още един тест с по- ниска стойност на параметъра learning rate.

Интерес буди и фактът, че в началните епохи, точността при валидация е по-висока от тази при трениране. Това се случва поради две причини:

**1. Data augmentation/dropout**- Тези техники за намаляване на овърфитинга са активни само по време на тренирането(но не и по време на валидацията). Тоест, невронната мрежа изкуствено е затруднена при обучението, което води и до по-лоши стойности на точността. Въпреки това, тя ще генерализира по-добре.

**2. Оценката на резултатите при трениране/валидация**- към момента, реализацията на tensorflow(в частност keras) оценява резултатите при трениране и валидация по различен начин. При трениране, резултатите се изчисляват per-batch- в конкретния случай на всеки 32 изображения, защото batch size=32. След това крайният резултат се получава чрез средно аритметично на стойностите за отделните batch-ове. В първите от тях, мрежата не се справя добре и това води до ниска batch accuracy и респективно висока batch loss. След усредняване на стойността на база всички batch-ове от епохата, то лошите стойности притеглят средната стойност надолу. Друг е начина на изчисление при валидацията- използват се крайните тегла(в случая най-добрите), получени в епохата след трениране, което води до по-добри стойности на валидацията в началото на обучителния процес.

**Тест 3.**

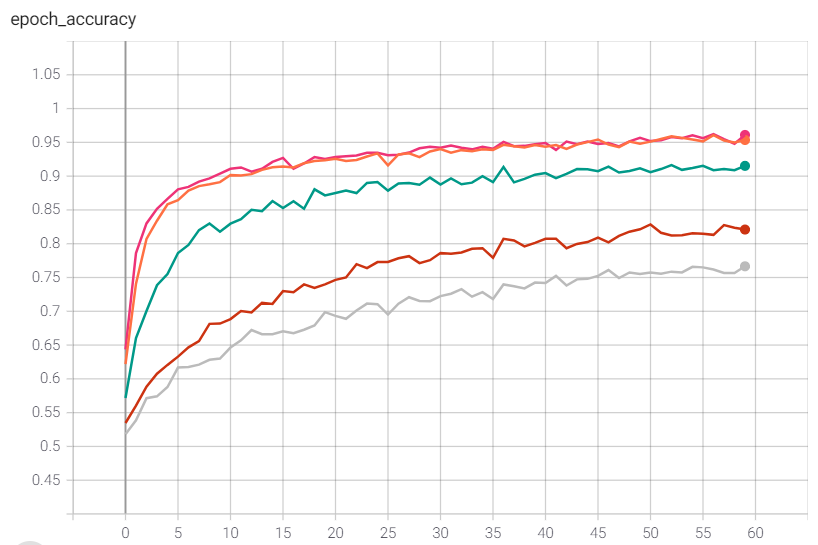
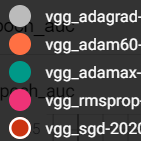
В предните 2 теста е използван един от най – разпространените и съвременни оптимизационни алгоритми- Adam(adaptive optimizer), доказано даващ много добри резултати при трениране на невронни мрежи. Съществуват обаче и други видове оптимизационни алгоритми. Подбирането на оптимизатор е важна част от обучителният процес. Затова в този тест ще бъде проверено кой оптимизатор е най-подходящ за текущия набор от данни и респективно конкретната НМ. Използваната архитектура е като тази в тест 2, а използваният Learning Rate е 1.10-6.

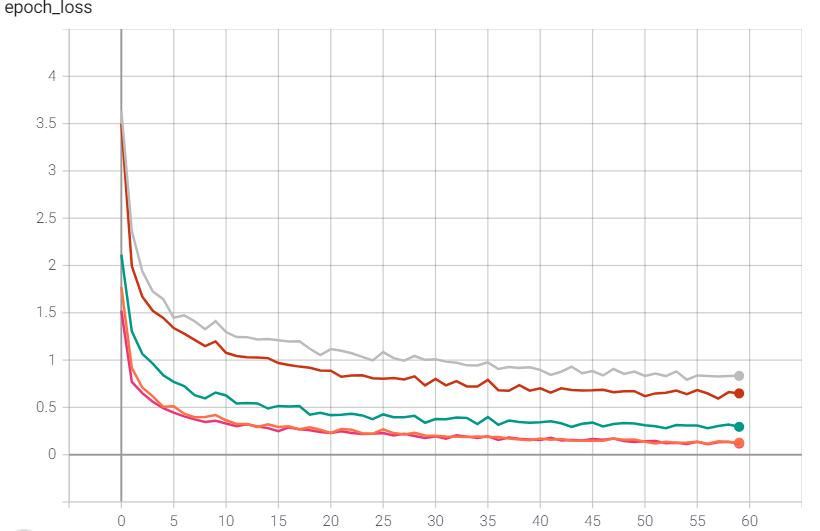
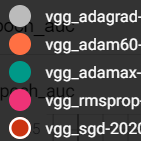
Тествани оптимизационни методи:

* Adam
* RMSProp
* Adagrad
* Stochastic Gradient Descent
* Adamax

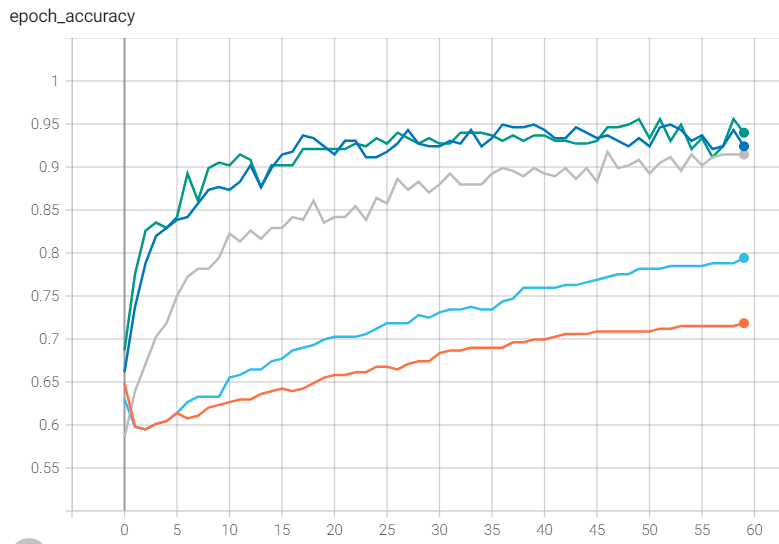
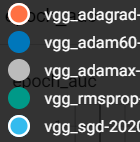
След трениране за 60 епохи се получават следните резултати:

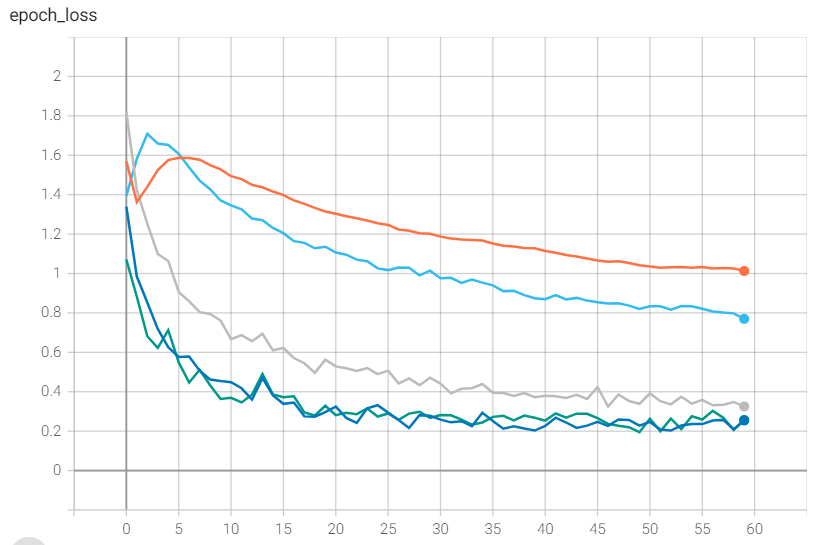
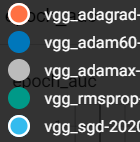
За трениращите точности и loss функция:





За валидационните точност и loss функция:





Както се вижда от графиките, най- зле се представят SGD и Adagrad методите, а най- добре- RMSProp и Adam. Обучението с Adam и RMSProp е чувствително по-бързо, като около 25-та епоха стойността на loss функцията е сведена до 0,2. Друга особеност, която прави впечатление в графиките е повишеното осцилиране при някои от методите- Adamax, RMSProp и Adam- това се дължи на динамичното адаптиране на learning rate(LR) в процеса на обучение, защото тези оптимизатори са адаптивни(за разлика от SGD например). Забелязва се още, че осцилирането е по- силно в началните епохи- това е така, защото тогава са по-големи промените в LR и съответно в стойностите за теглата.

Разбира се, всеки оптимизационен алгоритъм изисква различно настройване на хиперпараметрите си, но може да се обобщи, че за текущият разглеждан проблем Adam и RMSProp изискват минимални допълнителни настройки и достигат сравнително бързо оптималното или близко до оптималното решение. Кривата на loss функцията за SGD намалява много плавно, което се дължи на ниската стойност на learning rate параметъра. SGD най - често се използва с по-висока стойност на LR и поетапното му намаляване. Това става обикновено под формата на LearningRateScheduler, намаляващ LR по предварително дефинирана от разработчика схема. Предимствата на Adam оптимизатора са :

* Лесен за използване
* Не се нуждае от много настройки
* Адаптира самостоятелно Learning Rate параметъра и поддържа различни LR за различните параметри
* Изисква по-малко памет- спрямо по- старите методи като SGD
* Подходящ за градиенти с високо ниво на шум
* Изчислително ефикасен
* Бърз

На база горните графики за следващите тестове отново ще бъде използван Adam оптимизатора, предвид добрите резултати, които се постигат и множеството му предимства.

**Тест 4.**

Тук ще бъде използвана същата архитектура, както в тест 2, но скоростта на обучение ще бъде намалена на 1. 10-6. Очакваният резултат е намаляване на осцилирането. Параметри на тестването:

Folder: vgg\_5

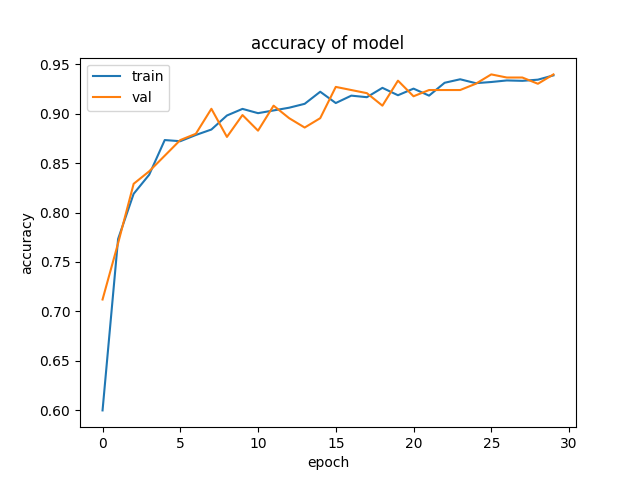
Image size: 128x128x3

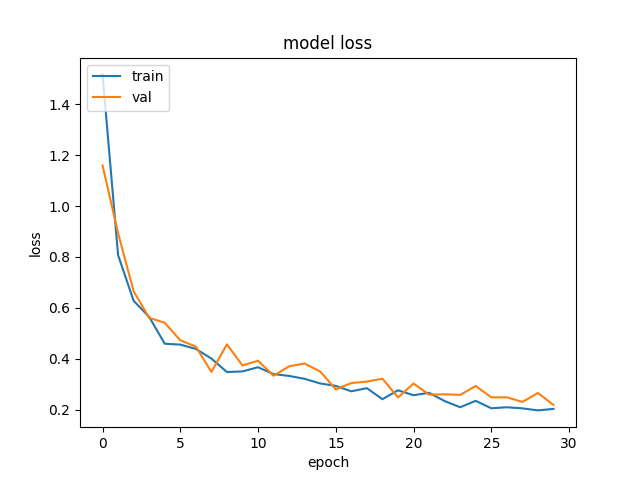
Batch size: 32

Epochs: 30

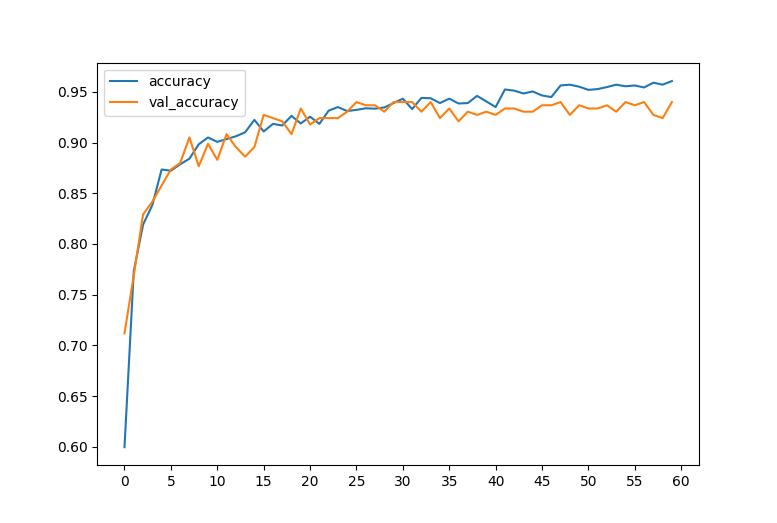
Learning rate: 0.000001 (1.10-6)

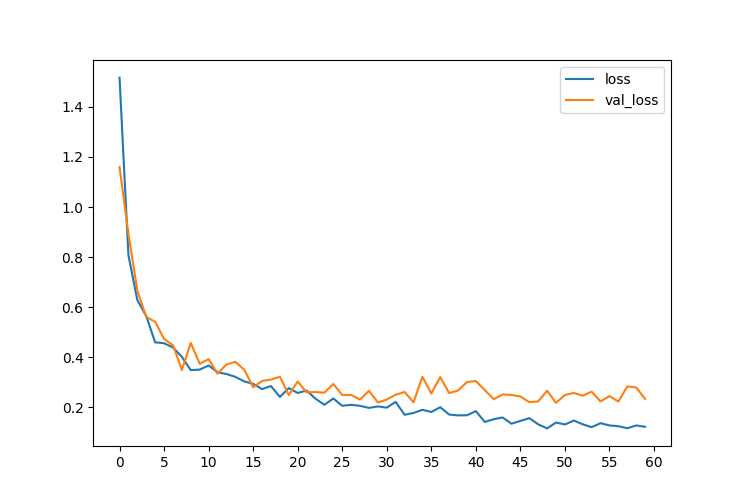
Резултати:





От графиките се вижда намаляване на осцилирането и стабилизиране на кривите. В продължение на обучението се вижда едновременно подобряване както на валидационните, така и на трениращите резултати. Стойностите при валидация/трениране са много близки - кривите са почти една върху друга. Максималната достигната точност е около 93%. Може да се каже, че мрежата генерализира добре и дори позволява още обучение, тъй като не се наблюдава спад/стагнация при валидирането. Следва трениране за още 30 епохи:





Вижда се, че оптималната достигната точност е 93,98% в епоха 34. В следващите епохи, валидационните резултати започват да се различават от тези при трениране и съответно модела започва да се пренагажда. Зоните, където се случва това са оградени със зелен цвят. По-нататъшно трениране на **тази архитектура** на НМ с **тези стойности** на хиперпараметрите няма да доведе до по-добри резултати, а напротив- мрежата започва да разчита прекалено много на train сета и няма да се справи добре с невиждани досега данни(което личи и от валидационната крива след епоха 35).

Тенденцията за подобряване на резултатите при трениране навеждат на мисълта, че мрежата „има какво да научи още“- кривите на точността и loss функцията се подобряват при трениране, вместо да се наблюдава насищане/спад. В този случай единственото ограничение, налагащо спиране на обучаващия процес е overfitting-ът на модела. Възможни са два подхода, за да се повиши точността при валидация и да се извлече маскимума от данните, без да се наруши генерализацията. Тези два подхода са- опростяване на архитектурата и прилагане на регуляризация.

**Подход 1 - Опростяване на архитектурата.**

Възможна причина за пренагаждане на модела е прекалено сложната архитектура на невронната мрежа. В оригиналната VGG16 имплементация, като top слоеве са използвани 2 плътносвързани слоя с **4096** неврона. В тренирано в този опит като крайни слоеве са използвани също 2 dense слоя, но с **2048** неврона. Следва да се провери възможно ли е намаляване/забавяне на пренагаждането напред във времето чрез редуциране броя на невроните в тези слоеве. Така архитектурата придобива следния вид:

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(1024, activation="relu")(val)  
val = krs.layers.Dropout(0.1, seed=1333)(val)  
val = krs.layers.Dense(1024, activation="relu")(val)  
outputs = krs.layers.Dense(1, activation='sigmoid')(val)  
return krs.Model(inputs, outputs)

Параметри на теста:

Folder: vgg\_6

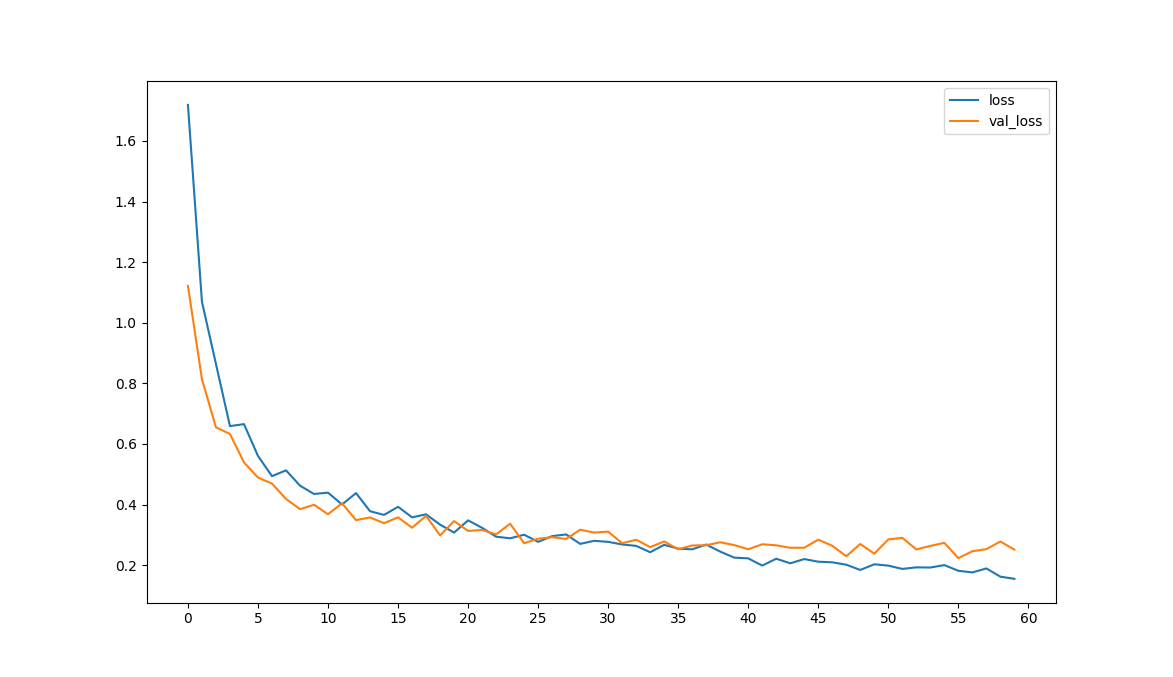
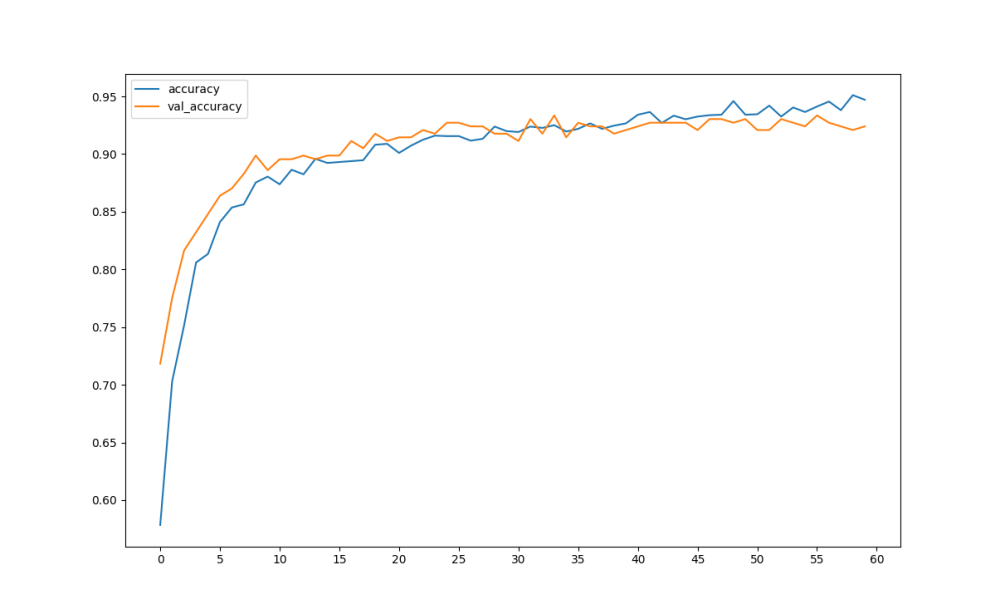
Image size: 128x128x3

Batch size: 32

Epochs: 60

Learning rate: 0.000001 (1.10-6)

Резултати:



При извършеното трениране за 60 епохи се вижда, че достигнатата максимална валидационна точност е 93,35%, а минималната стойност на ценовата функция- 0.229. След епоха 48 мрежата започва да се пренагажда спрямо трениращия сет. Може да се заключи, че направената редукция на броя неврони в top слоевете на мрежата от 2048 на 1024 не спомага намаляването/забавянето на overfitting-a, а дори максимално достигнатата валидационна точност е малко по-ниска(93,35 спрямо 93,98%). Следователно, опростяването на мрежата не носи ползи за конкретния експеримент.

max val acc- 93,35% в епоха 34

min val loss – 0.229 в епоха 48

**Подход 2 - Прилагане на регуляризация**

Регуляризацията е метод за справяне с пренагаждането на различни ML модели към трениращия сет. Тя бива два вида – L1 и L2. В естеството си, регуляризацията се състои в свиване големината на теглата на параметрите, резултиращо в намаляване на сложността на модела. L1 и L2 регуляризациите се различават по това, че L1 позволява свиване на теглата до 0(водещо до премахване на чертите, които мрежата счита за маловажни), докато L2 не позволява свиване до 0(т.е не се губи информация).

В този случай ще бъде използван по-малко рестриктивния вариант L2. Регуляризацията ще се прилага само върху изходните top слоеве(Dense с 2048 неврона), тъй като в експеримента се използва техниката transfer learning и теглата на feature extractor слоевете са замразени. Отразяване на регуляризацията в кода на архитектурата:

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.GlobalAveragePooling2D()(val)  
val = krs.layers.Flatten()(val)  
val = krs.layers.Dense(2048, activation="relu",  
 kernel\_regularizer=krs.regularizers.l2(0.000001))(val)  
val = krs.layers.Dropout(0.1, seed=1333)(val)  
val = krs.layers.Dense(2048, activation="relu",  
 kernel\_regularizer=krs.regularizers.l2(0.000001))(val)  
#val = krs.layers.Dropout(0.3)(val)  
outputs = krs.layers.Dense(1, activation='sigmoid')(val)  
return krs.Model(inputs, outputs)

Параметри на теста:

Folder: vgg\_v8reg

Image size: 128x128x3

Batch size: 32

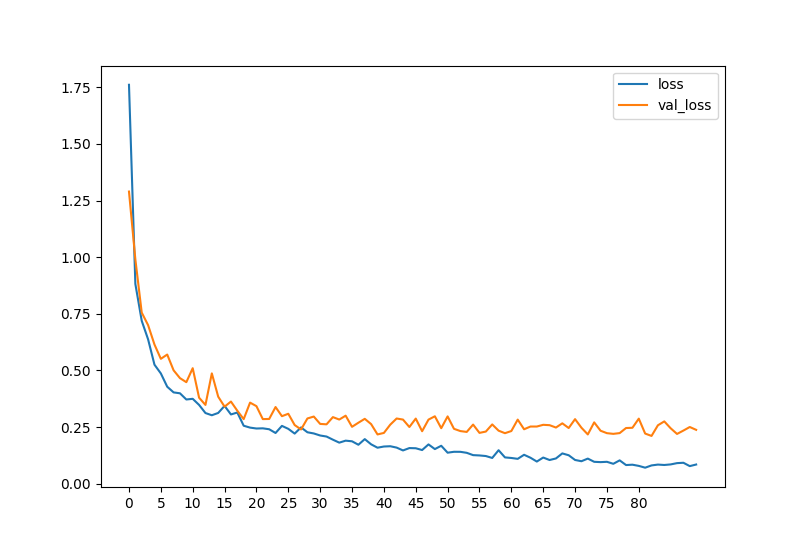
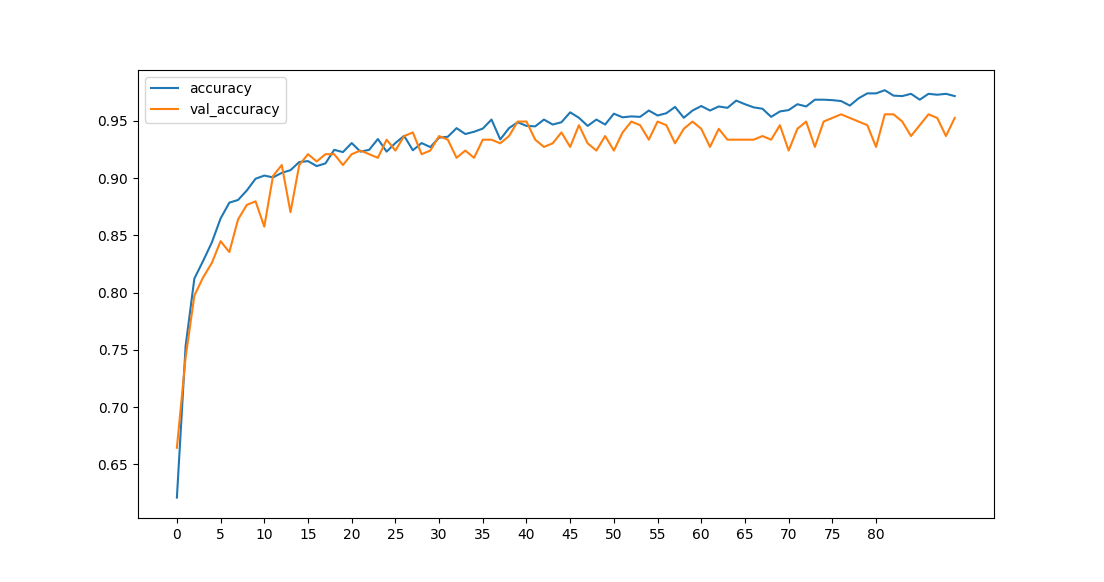
Epochs: 90

Learning rate: 0.000001 (1.10-6)

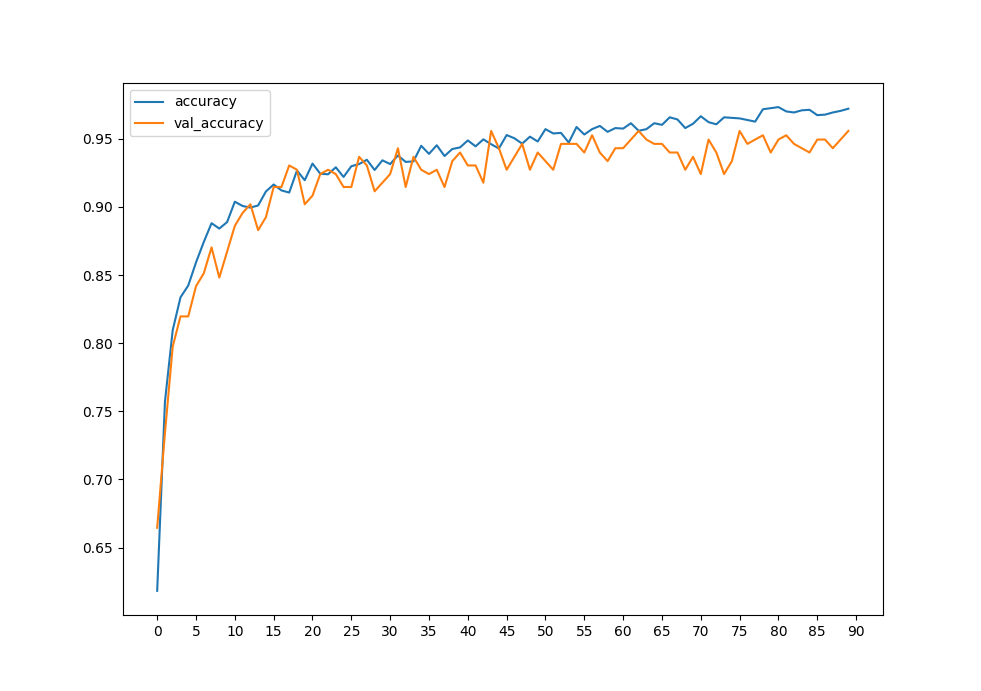
Optimizer: Adam

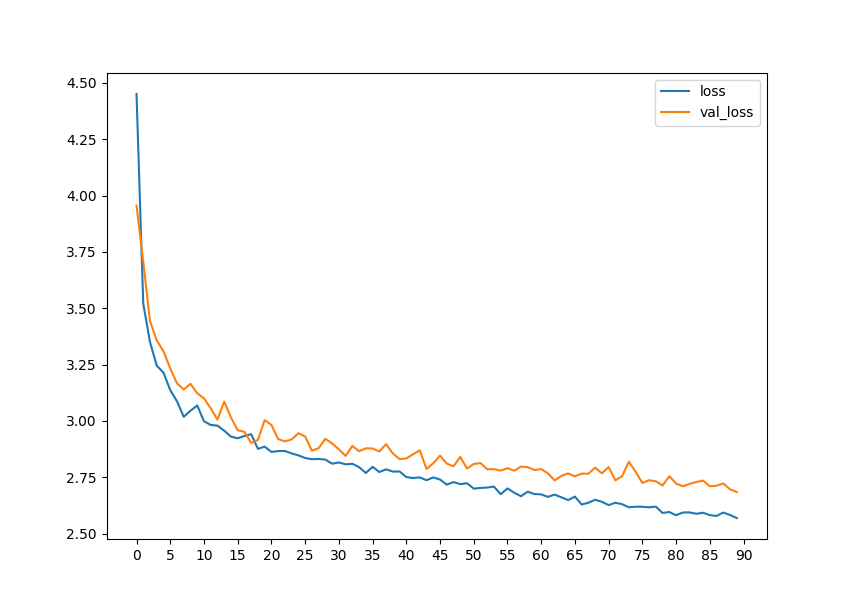
L2 regularization rate: 0.000001

Резултати:



Както личи от графиките, максималната валидационна точност е 94,93% епоха 40. След това не личи особено подобрение в кривата на точността, а в кривата на ценовата функция се вижда стагнация/увеличаване. Сравнявайки графиките на тестване с и без регуляризация, не се забелязва особено голяма полза от регуляризирането. За теста бе използвана ниска стойност на L2, която видимо все още не ограничава достатъчно невронната мрежа в обучението ѝ, следователно тя трябва да бъде увеличена. Затова ще бъде променена регуляризационната стойност на 0.0005. Тази стойност е предложена и от Франсоа Шоле(създателят на Keras) за мрежата Inception, тренирана върху ImageNet дейтасета. След ново трениране за 90 епохи, този път с регуляризация 0.0005 се получават следните графики:





Този път се вижда по-висока достигната стойност на валидационната точност- 44 епоха – 95,56%. Тази стойност е с около 2% по-голяма, от колкото достигнатата при обучение без регуляризация. След епоха 63 се наблюдава пренагаждане на модела. Интерес буди повишената стойност на ценовата функция. В останалите тестове, още след първите няколко епохи тя е сведена под 1, като насищане се наблюдава около 0,2. В този случай обаче, тя има доста по- високи стойност- над 2,5. Това е така именно заради приложената регуляризация, непозволяваща на мрежата да наподоби прекалено много трениращия сет и да се получи овърфитинг.

Обобщение на резултатите от регуляризацията:

|  |  |  |
| --- | --- | --- |
| Параметри    Величина | Max тренираща точност(%) | Max валидационна точност (%) |
| Без L2 регуляризация | 94,36 | 93,99 |
| L2= 0.000001 | 94,87 | 94,93 |
| L2= 0.0005 | 95,26 | 95,96 |

Приложената регуляризация доведе до желания резултат- наблюдава се намаляване на пренагаждането, позволяващо по-продължително трениране и повишаване на валидационната точност.

Регуляризацията трябва да бъде прилагана внимателно, защото при прекалено висок коефициент, тя може да доведе до underfitting. Затова е нужно да се намери подходящата стойност, която хем да намали overfitting-a и да може да се извлече по- висока точност, хем да не доведе до underfitting и алгоритъмът да се справя зле и с трениращите данни(да се понижи точността). В този случай, достигнатата точност от 95,56% е достатъчно задоволителна и не се налага промяна на регуляризацията.

**Опит 2.**

В този опит ще бъде използвана собствена имплементация на невронна мрежа, базирана на Inception модула. Идеята на този модул е да се разграничи частично от традиционните дълбоки невронни мрежи. В него се извършват паралелно конволюции с филтри с различни размери(най-често 1x1,3x3,5x5), след които резултатите се обединяват и подават на следващият такъв модул. Съществуват различни видове Inception модули, спрямо различните версии на Inception мрежата.

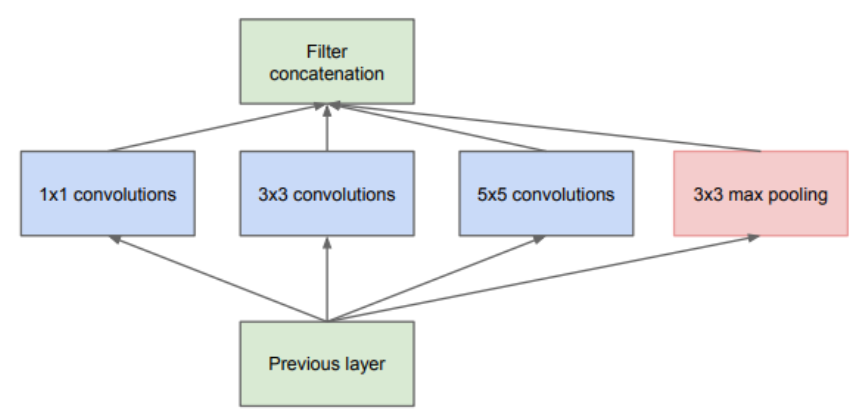
Тук ще бъде използвана традиционна версия на Inception модула, използвана в мрежата GoogLeNet, разработена от Гугъл през 2015г. Той включва:

* Конволюция с филтри 1x1 с relu активация и нулев падинг
* Конволюция с филтри 3x3 с relu активация и нулев падинг
* Конволюция с филтри 5х5 с relu активация и нулев падинг
* Макс пулинг слой

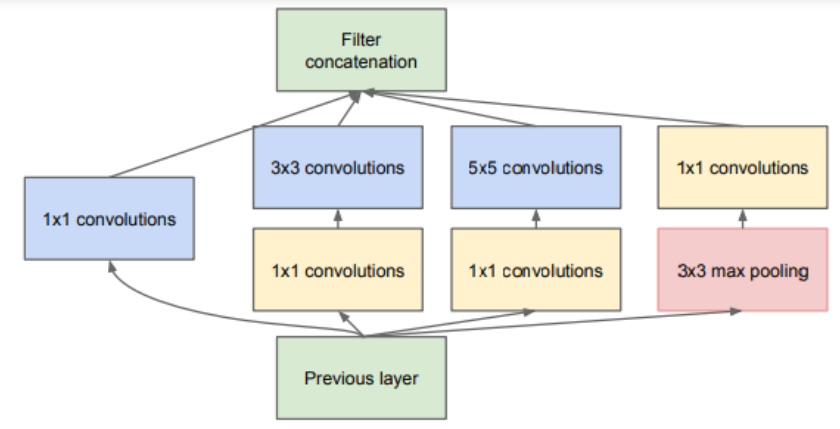
Код на Inception клетката:

def my\_inception(layer\_input, x1, x2\_reduction, x2, x3\_reduction, x3, x4):  
 # 1x1 conv  
 conv1 = krs.layers.Conv2D(x1, (1, 1), padding='same', activation='relu')(layer\_input)  
 # 3x3 conv  
 conv3 = krs.layers.Conv2D(x2\_reduction, (1, 1), padding='same', activation='relu')(layer\_input)  
 conv3 = krs.layers.Conv2D(x2, (3, 3), padding='same', activation='relu')(conv3)  
 # 5x5 conv  
 conv5 = krs.layers.Conv2D(x3\_reduction, (1, 1), padding='same', activation='relu')(layer\_input)  
 conv5 = krs.layers.Conv2D(x3, (5, 5), padding='same', activation='relu')(conv5)  
 # 3x3 max pooling  
 pool = krs.layers.MaxPooling2D((3, 3), strides=(1, 1), padding='same')(layer\_input)  
 pool = krs.layers.Conv2D(x4, (1, 1), padding='same', activation='relu')(pool)  
 # concatenate filters, assumes filters/channels last  
 layer\_out = concatenate([conv1, conv3, conv5, pool], axis=-1)  
 return layer\_out

Така наречената „наивна“ имплементация прилага директно конволюция на филтрите с различните размерности в следния вид:

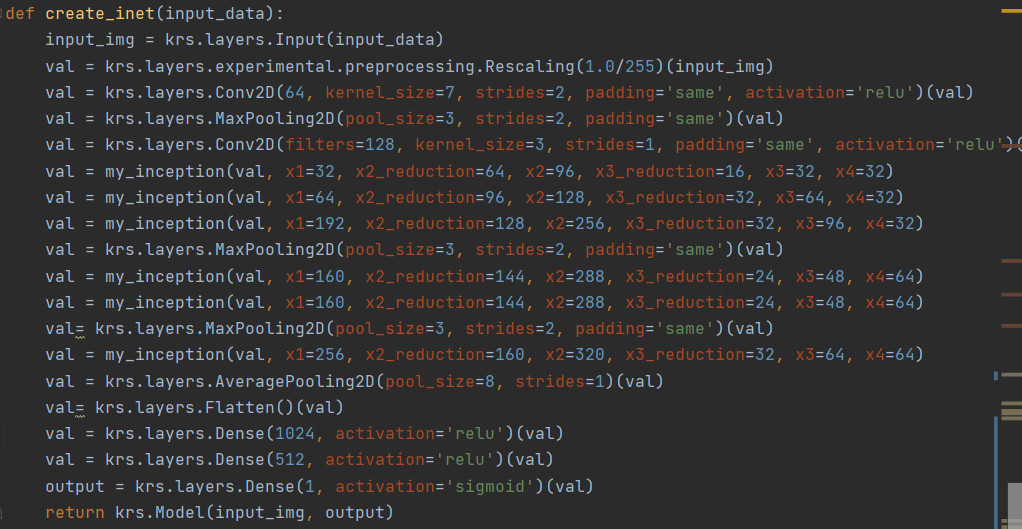


Това обаче води до необходимостта от поддържане на много голям брой параметри, водещ до по-бавно и изискващо много ресурси трениране на мрежата. Именно затова, авторите на модула предлагат вариация, използващ конволюция с филтри с размер 1х1 преди прилагането на останалите филтри. Това намалява размерността, респективно и параметрите. Именно тази вариация е използвана и в текущата имплементация:



Този модул е използван в имплементацията на мрежата, като са подавани различни стойности за броя на различните филтри. Важно е да се отбележи, че архитектурата на невронната мрежа по-долу е началната такава(т.е. subject-to-change) и върху нея ще бъдат прилагани подходящи промени на база постигнатите резултати от трениращият процес.

Код на архитектурата:



Както се вижда от кода, стойностите на пикселите на входните изображения са скалирани в интервала [0;1]. Преди прилагане на Inception модула са извършени 2 конволюции и MaxPooling операция за екстракция на основните черти на изображението. След приложените Inception блокове е извършен AveragePooling за намаляване на размерността и екстракция на най-важните черти. Flatten слоя извършва подготовката на данните за подаването им на плътносвързаните Dense слоеве със съответно 1024 и 512 неврона. Като изходен слой е използван Dense с 1 неврон и сигмоидна активационна функция, свиваща резултата в рамките на 0 и 1, отразяващ принадлежността към съответния клас. За разлика от Опит 1, където техниката на трансферирано обучение изискваше използване на 3-канални изображения, в този опит ще бъдат използвани изображения в режим **на сива скала с 1 канал**.

**Тест 1.**

Извършено е трениране за 50 епохи с гореописаната архитектура. Както се вижда, към мрежата не са приложени никакви мерки за регуляризация, намаляване на овърфитинга и обогатяване на данните.

Параметри на теста:

Learning rate: 0.0001

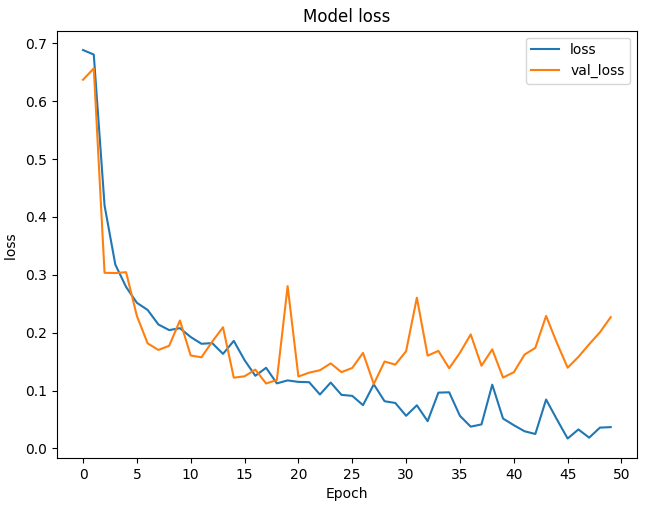
Optimizer: Adam

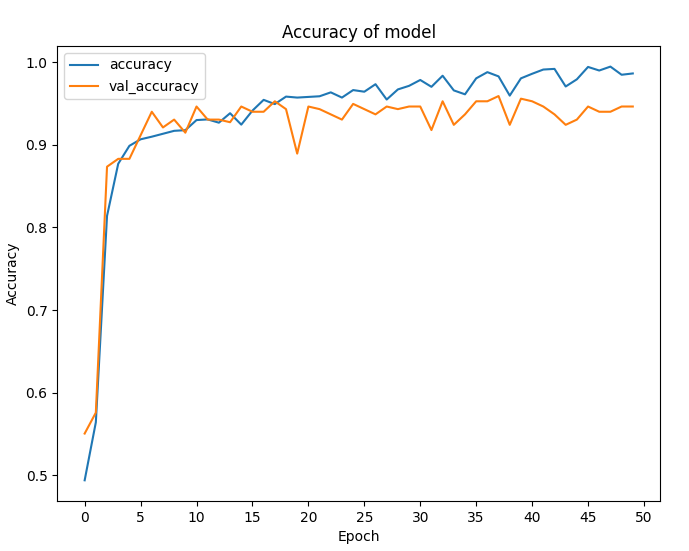
Image size: 128x128x1

Batch size: 32

Epochs: 50

Резултати:





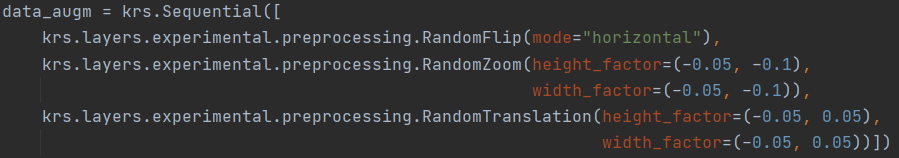
От графиките се вижда, че в процеса на обучение трениращата точност нараства, а стойността на ценовата функция намалява- мрежата се обучава успешно. Началната точност е около 50%, което в контекста на машинното обучение и изкуствения интелект се определя като “random guessing”- т.е. мрежата случайно „налучква“ прилежащия клас. Достигната точност при трениране е висока – около 98%. Добра тенденция се наблюдава и при валидационните резултати(оранжевата крива на графиките), но тя продължава до 18 eпоха. След това ясно се вижда влошаване/стагнация на резултатите при валидация, паралелно с подобряване на трениращите такива- моделът започва да се пренагажда и да се справя зле с невиждани досега данни. Достигнатата максимална валидационна точност е 95,25 % в епоха 18. В идеалният случай, кривите на валидация и трениране трябва да бъдат една върху друга. Това, разбира се, е трудно възможно, затова като допустима разлика за целите на експеримента може да се определи 0.5-1%.

Продължаващото повишаване на точността при трениране навежда на мисълта, че ако бъде намален овърфитинга, валидационната точност също ще се увеличи. Затова към архитектурата от този тест ще бъдат добавени регуляризационни техники, като dropout и обогатяване на данните.

**Тест 2.**

В този тест ще бъдат приложени техники за намаляване на пренагаждането на мрежата спрямо данните за трениране с цел да се постигне по-добра генерализация и по-висока точност при невиждани досега изображения.

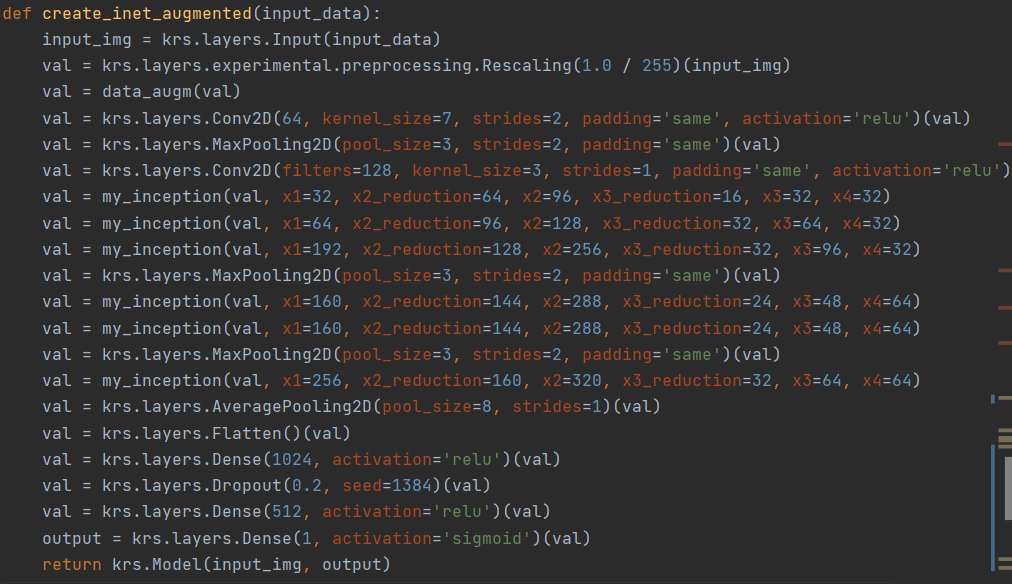
Първата такава техника е обогатяването на данните- произволни ротации, транслации и зуум на входните изображения. В текущия случай това са:

****

Това обогатяване повишава устойчивостта на модела, повишава разнообразието на входните данни и по този начин значително намалява пренагаждането.

Втората такава техника е Dropout слоят, който занулява част от входната информация, като по този начин невронната мрежа се научава да не разчита прекалено много на някои открити черти в изображенията.

След прилагане на описаните методики, архитектурата придобива следния вид:

****

Следва трениране на новата архитектура със следните параметри:

Learning rate: 0.0001 папка inception\_v2

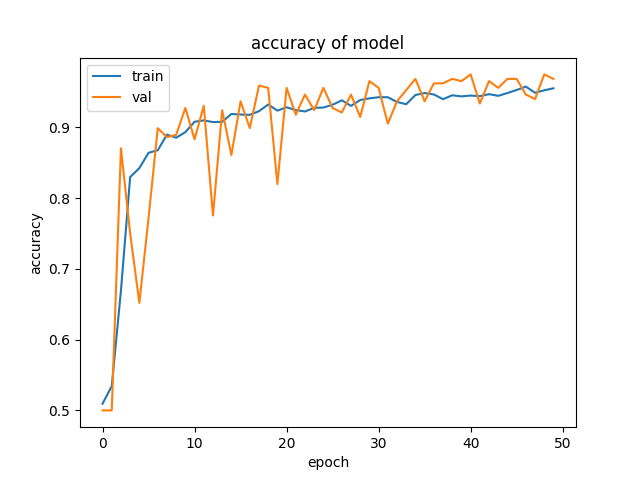
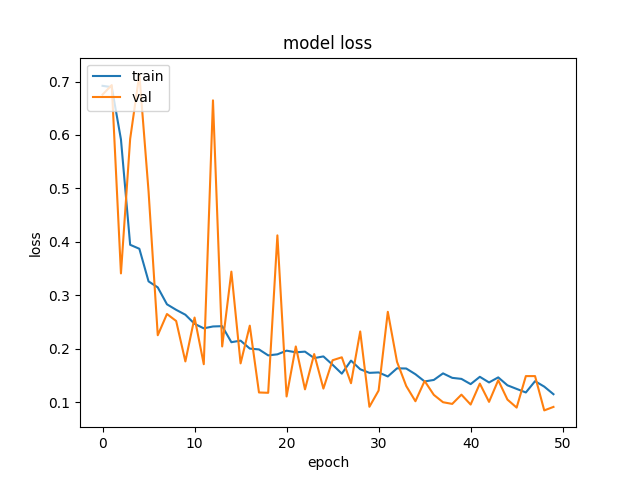
Optimizer: Adam

Image size: 128x128x1

Batch size: 32

Epochs: 50

Резултати от тренирането:

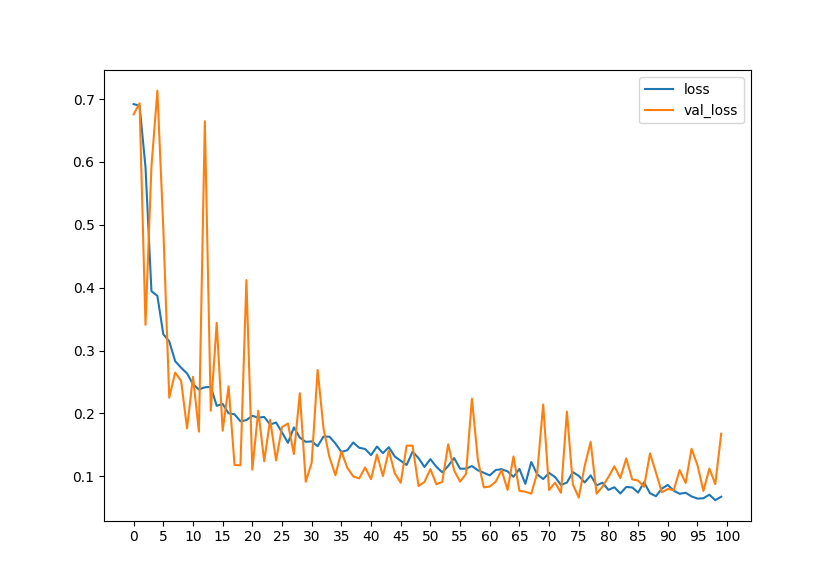


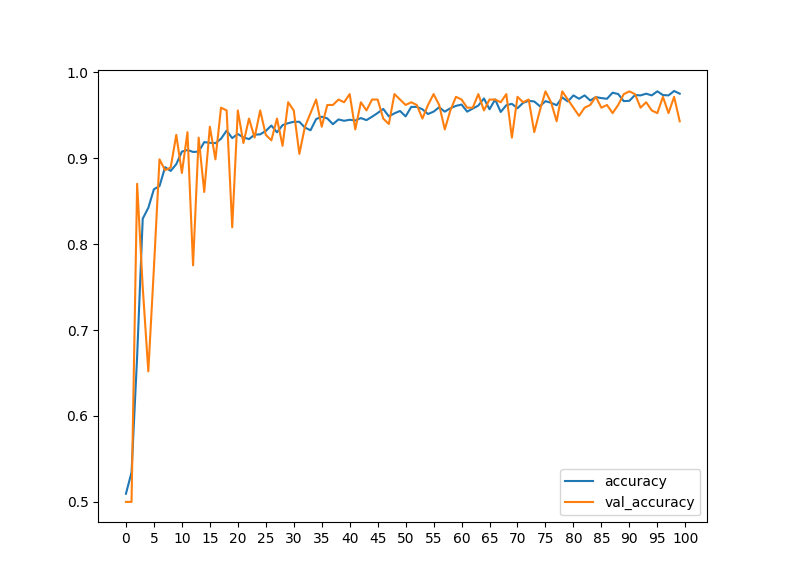
Достигнатата максимална валидационна точност е 97,47%, а трениращата 95,74%, като не се наблюдава пренагаждане на модела. В графиките впечатление прави силното осцилиране на кривите на валидационния сет. То е по-изявено в началото на трениращият процес(интервала 1-25 епоха), защото тогава промените в теглата са по-големи. За това допринася и Adam оптимизатора, който използва адаптиращи се отделни тегла за отделните параметри. Осцилирането намалява след епоха 35, защото промените в теглата и Learning rate параметрите стават все по- плавни- доближава се оптималното решение. Напред във времето на трениране се очаква още по-осезаемо стабилизиране на валидационните стойности.

Тук се наблюдават на места по- добри резултати при валидация, от колкото при трениране, за разлика от предния тест, където валидационните резултати бяха традиционно по-лоши. Това се дължи главно на 2 неща:

* Изкуствено затруднената мрежа при трениране- **data augmentation** слоя и **dropout** слоя са **неактивни** по време на валидация. Тогава мрежата използва пълният си капацитет, при това на необогатени данни, които се очаква да класифицира по-лесно, защото не са изкуствено зашумени. Тези разлики във валидиращият/трениращият процес водят до по-добри резултати при валидация.
* По - малкият брой изображения за валидация- тренирането се извършва с 2536 изображения, докато валидацията се извършва с 316.

За да бъде направена качествена оценка на архитектурата е необходимо тренирането да продължи. Следва обучение за още 50 епохи:





Най-високата достигната валидационна точност е 97,77% в епоха 78. В резултатите се забелязва спад на валидационната точност/увеличаване на стойността на ценовата функция след епоха 80- оградената със зелена елипса част. Може да се заключи, че мрежата е започнала да губи своята генерализация. Процесът на трениране на текущо разглежданата архитектура се прекратява, тъй като той не би довел до по- добри резултати.