MACHINE LEARNING
REPORT 5

****The jupyter notebook with the code is at the end of the report***

I. INTRODUCTION

i. BACKGROUND & MAIN OBJECTIVE

I will work with a dataset of images of brain tumors. The data is subdivided in 4 different types of tumors and the goal is to recognize which type of brain tumor is present in the image. This is a job that is currently being done by the doctors at the clinics, however, here I will try to make a machine do it through the implementation of a Convolutional Neural Network. Hence this is a classification problem. As you will see throughout this report it hasn't been an easy journey and some problems I was not able to solve. Even though the result is not as I was expecting it has been an instructive journey.

II. THE DATA

i. DESCRIPTION OF THE DATASET

The data is comprehensive of 3264 grayscale brain tumor pictures that are categorized in 4 classes:

- Glioma Tumor
- Meningioma Tumor
- No Tumor
- Pituitary Tumor

The dataset is subdivided in 2870 training images and 394 test images. Which is a 87% - 13% of the whole dataset.

ii. DATA GATHERING & FEATURES ENGINEERING

First of all I needed to collect all the data and find a way to store them in an array in a way that the keras convolutional neural network was able to work with. Do to this I made use of the OpenCV library to read the images, make sure all of them where in grayscale and resize them. The majority of the original pictures are 500x500 with some exceptions where the images where smaller which will cause problems when the list was needed to be converted in an array. Resizing all the images to 200x200 ensured that all pictures where the same size and reduced the amount of memory taken by the future algorithms. Subsequently I normalized all the images by dividing all the training and test sets by 255 ensuring that all arrays had numbers between 0 and 1. Lastly I one hot encoded the 4 brain tumor categories so instead of 0,1,2,3 I have a 1 in whatever category is attributed to the given image.

III. DEEP LEARNING METHODS

For starters I decided to implement a known convolutional neural network that I knew I was supposed to work. Therefore I used the same network used in the notebook. I called this method vanilla for future reference.

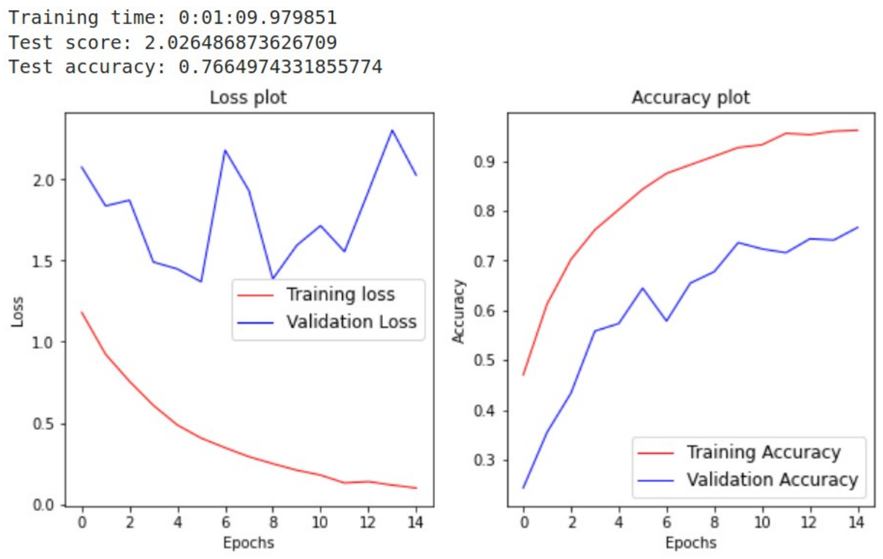
- **The Vanilla network**
  This scheme of this model is as follows:

  Conv2D → Conv2D → MaxPooling2D+DropOut → (Flatten) → Dense+DropOut → Final Classification

  The activation function used is ReLU throughout all the network. The filter depth is set to 32 and padding equal to ‘*same*’ as was done in the notebook Convolutional Neural Network, however, in this model I used a different kernel size and stride. Given the size of the images, which are 200x200 pixels, I decided to use a bigger kernel of 10x10 and a wider stride of 5x5. I made this choice in order to have a model that would train fast, not for sake of accuracy, but to test that there was no problem with the image arrays I handcrafted and to see a fist general behavior of a convolutional neural network on my dataset.

  In order to train the model I defined a function called ‘*m_train*’ which as a set batch size of 16, will train and shuffle on 15 epochs and calculates the validation loss and accuracy at each epoch. This function will also take note of the time that the model will take to train and will print out a plot showing the training loss vs. validation loss and accuracy loss vs. validation accuracy.

  The model has been complied using the ‘*categorical_crossentropy*’ loss function which should more logic choice since we are dealing with 4 different categories. The optimizer is set to RMSprop and the metrics has been set to ‘*accuracy*’.

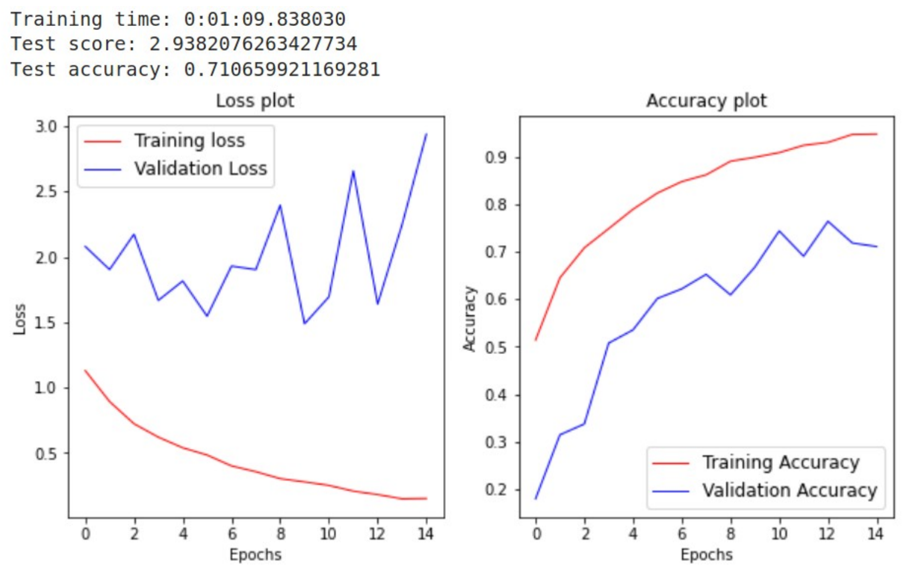


```
Training time: 0:01:09.979851
Test score: 2.026486873626709
Test accuracy: 0.7664974331855774
```

Figure 1. Plots of Loss and Accuracy changes to the training set and test sets for the CNN Vanilla model

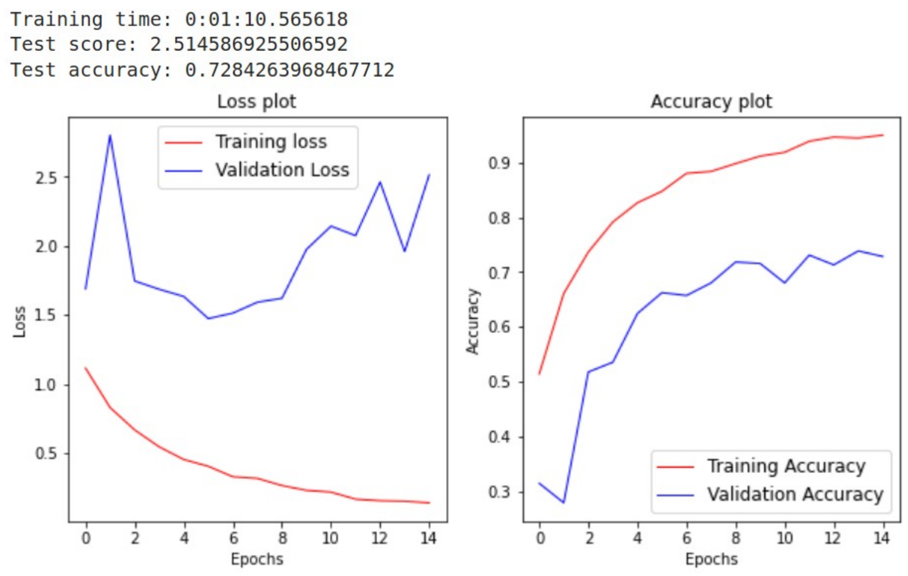
This model took a little more the 1 minute to train on 15 epochs, showing an accuracy on the test set of ~75% and a very high test score of more then 2. The validation score should be much lower. Such a high number as we have here means that the model is not able to generalize well. From the graph above, we see that the validation loss is not steadely decreasing but instead it seems to get higher as the model is being train. I wasn’t expecting a good results out of this model because, as previously said, it was just to test the waters. However it the behavior observe is troubling. Even though the model is fairly simple I still would have expected a lower validation score.

In order to find a more optimized model I started to probe different activation functions such as leakyReLU and tanh.



```
Training time: 0:01:09.838030
Test score: 2.9382076263427734
Test accuracy: 0.710659921169281
```

Figure 2. Plots of Loss and Accuracy changes to the training set and test sets for the CNN Vanilla model - LeakyReLU



```
Training time: 0:01:10.565618
Test score: 2.514586925506592
Test accuracy: 0.7284263968467712
```

Figure 3. Plots of Loss and Accuracy changes to the training set and test sets for the CNN Vanilla model - tanh

The different activation function doesn’t seem to solve any problems as the behavior over 15 epochs is still the same as the one observed when using the ReLU activation function. The use of the LeakyReLU and tanh seems to have worsen the problem showing and actual increase of the validation loss meaning using this functions has made the model learn even less.

The next hyper-parameter I tempered with is the batch size. Until now I used a batch size of 16 because I though that since I do not have much images in my database I could benefit from training more batches. However, it came to my mind that it is possible that too using batch sizes that are too small can prevent the model to learn. Therefore I tried using different batch sizes: 8, 16, 32, 64, 128, 256. Unfortunately, increasing the batch size had a negative effect on my prediction. The validation loss gets worse and worse. The best batch sizes where 16 and 32.

I followed with experimenting with the filter depth. I tested this hyper-parameter using the values 32, 64, and 128.

```
Training time: 0:06:36.578635
Test score: 2.4069085121154785
Test accuracy: 0.7461928725242615
```

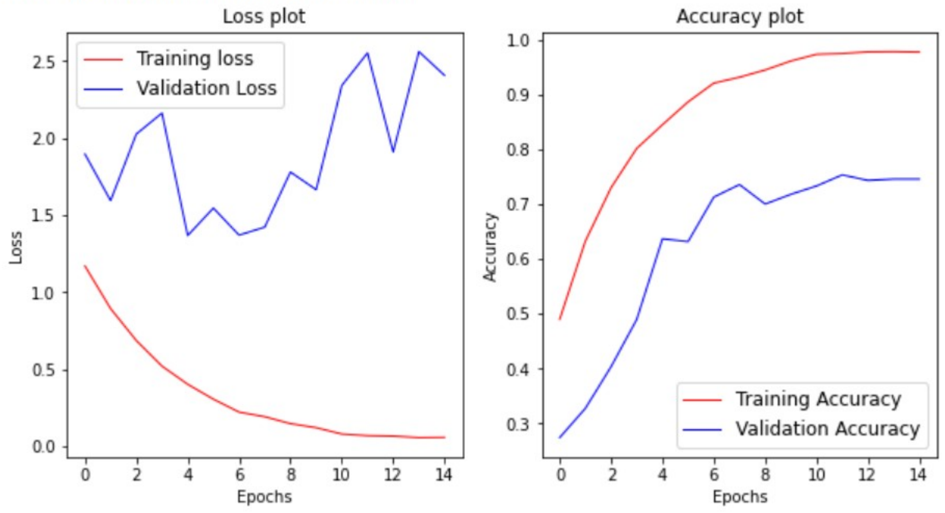


Figure 4. Plots of Loss and Accuracy changes to the training set and test sets for the CNN Vanilla model – Filter depth 128

As we can see changing the depth of the layers from 32 to 128 didn't change much we still go the same problem.

- **Convolutional Neural Network – Variation 1**

  Its time start changing the kernel and stride sizes. This will increase the training time by quite a bit but these are the last hyper-parameters that I can change. In order to do so I will need to build a more deep model in order to reduce the huge number of parameters that the model has to learn otherwise. The scheme of this network model is as follow:

  Conv2D → Conv2D → MaxPooling2D+DropOut → Conv2D → Conv2D → MaxPooling2D+DropOut → Conv2D → Conv2D → MaxPooling2D+DropOut → (Flatten) → Dense+DropOut → Final Classification

  This method will have to calculate almost 9 million parameters. The filter number is set to 32, the kernel size the 3x3 and the strides to 1x1. The padding has been set to '*same*' and '*valid*' alternately. The activation function used is ReLU throughout the whole model and '*categorical_crossentropy*' has been used as loss function.

```
Training time: 1:00:03.675333
Test score: 3.9023020267486572
Test accuracy: 0.7690355181694031
```

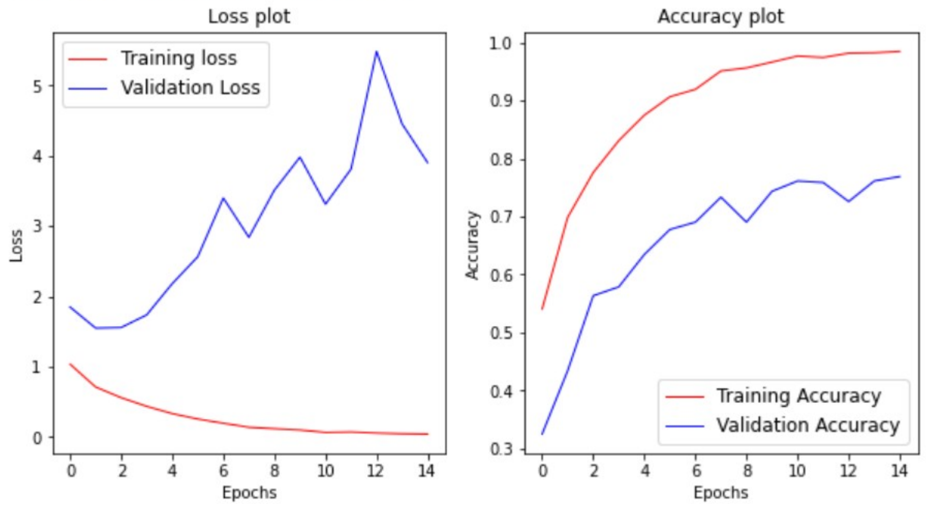


Figure 5. Plots of Loss and Accuracy changes to the training set and test sets for the CNN Variation 1

Unfortunately, even a more deep neural network didn't get it right. This model shows the highest ever test score with an accuracy on the test set of 76%. It took 1 hour to run. I tried also a similar convolutional neural network with less conv2d layers and more dense layers but I got the same results.

- **Convolutional Neural Network – Augmented**
  The problem on all the above models is that the validation loss is not improving as the model is being train. This means that probably the model is over-fitting and cannot generalize well on the test set and the accuracy seems to be capped at ~75%. There is one particular thing that I thought it could be a problem since I decided to work with this dataset, which is: the size of the dataset. If we recall from all the course notebook the dataset used where always comprehensive of tens of thousands of images. My dataset has only a little more the 3 thousand. For this reason I decided to try to use the ImageDataGenerator object which is built in Kersas. This allows to generate random images starting from the given dataset images by applying transformations on them. Therefore I proceeded on training my next model on batches of size 32 of random generated images. The model I used for this Neural Network has the same structure of the Vanilla model but with different kernel size (in this case 4x4), strides (in this case 2x2) and depth (in this case 32). As a reminder I report the model layers structure again here:

  Conv2D → **Conv2D** → **MaxPooling2D+DropOut** → **(Flatten)** → **Dense+DropOut** → **Final Classification**

  This model have almost 9.5 million parameters to be learned and will be trained over 25 epochs, will used the ReLU activation function and will be compiled with the '*categorical_crossentropy*' loss function.

```
Training time: 0:11:57.812785
Test score: 1.4453654289245605
Test accuracy: 0.1878172606229782
```

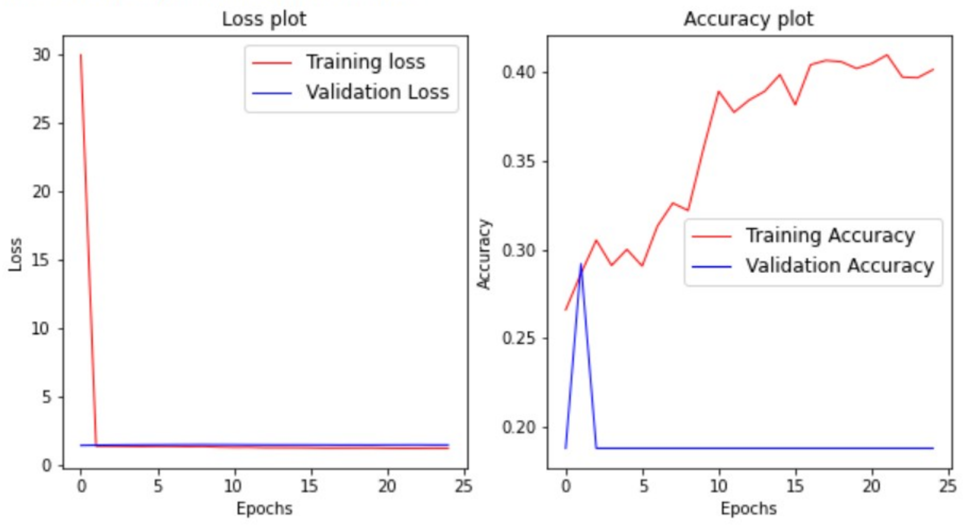


Figure 6. Plots of Loss and Accuracy changes to the training set and test sets for the CNN Augmented

This model is the worst model so far. Apparently generate random images for this specific dataset is not helpful. I believe that the images are all somewhat similar being all brain tumors which differ only by position in the image. Sometimes they have similar sizes or shapes. Moreover, the images are not always taken from the same angle which makes also the position of the tumor not very significative. This is probably way training a convolutional neural network on this dataset has been difficult. In this case, not only the validation loss and accuracy are at a low time worse but also the accuracy and loss on the train set is terrible. Using the random Image generator is the worst choice I could have made.

IV. ANALYZING THE RESULTS SO FAR

After playing around with many hyper-parameters, which non of the solved my problem, I started to think about how I compiled all the models. As it turns out, I have been always compiling them using the '*categorical_crossentropy*' as loss function. After looking into this loss function I came to realize that when '*categorical_crossentropy*' loss is used for classification, what happens is that bad predictions are going to be penalized much more than good predictions are rewarded. Which means that, even if many images are correctly predicted, a single misclassification will have a huge loss which will significantly increase the mean loss. In almost all my models accuracy and loss were increasing, which I think it means that the network is overfitting. Which leads to hugely wrong predictions on some validation images. However, even though the validation loss is increasing the validation accuracy increase as well. Which is counterintuitive but I think it is due to the fact that, besides all, the model is still learning some patterns which are useful for generalization. Moreover, in our multi-class case, where we have more then a binary classification the effects can be even more complex, since the network can be overfit on different class at each given epoch.

I will try to use the MeanSquaredError loss function. I don't really know if it should or shoudln't be used in my case. However I think this will help reduce the penalization for miss-classifications making the model learn better.

- **Vanilla model – MeanSquaredError loss Function**

  I will train the vanilla model over 50 epochs and with a batch size of 32. This should be fairly fast and will help us see if the change in the loss function used will give us better results.

```
Training time: 0:03:28.384329
Test score: 0.11975941807031631
Test accuracy: 0.7284263968467712
```

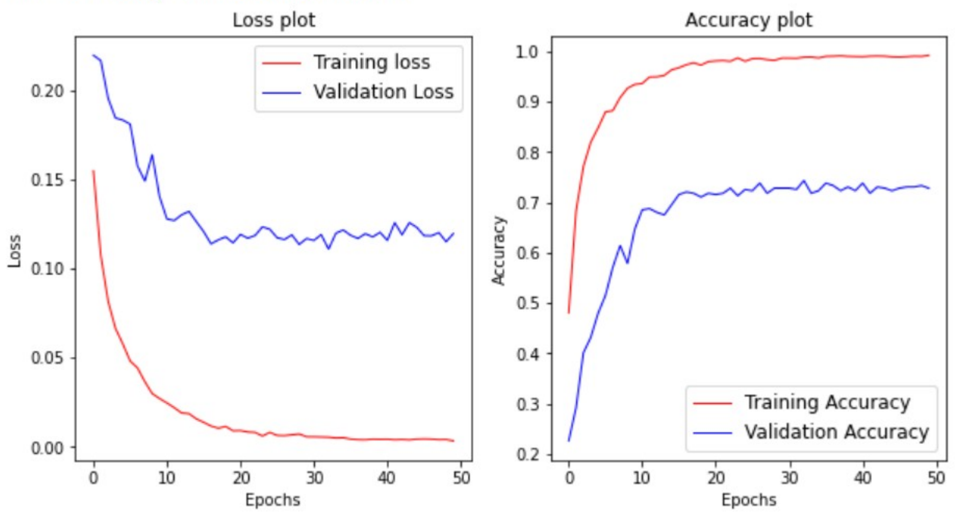


Figure 7. Plots of Loss and Accuracy changes to the training set and test sets for the CNN – MeanSquaredError loss function

Far from optimal but better then the previous models where the loss function used was the '*categorical_crossentropy*'. The validation loss is greatly decreased more the 10 folds. The model is still difficult to train and doesn't seem to be able to pass the 75% accuracy on the validation accuracy. However, this is a very simple model, lets see if a more deep network can give use better accuracy.

- **Convolutional Neural Network – Variation 2**

  This convolutional neural network has the following layer structure:

Conv2D → Conv2D → MaxPooling2D+DropOut → Conv2D → Conv2D → MaxPooling2D+DropOut → (Flatten) → Dense+DropOut → Final Classification

This model has a kernel size of 5x5 and stride 2x2 with padding equal to '*same*'. The activation function used is ReLU and it is compiled with the MeanSquaredError loss function.

```
Training time: 0:24:10.233510
Test score: 0.1388760507106781
Test accuracy: 0.6979695558547974
```

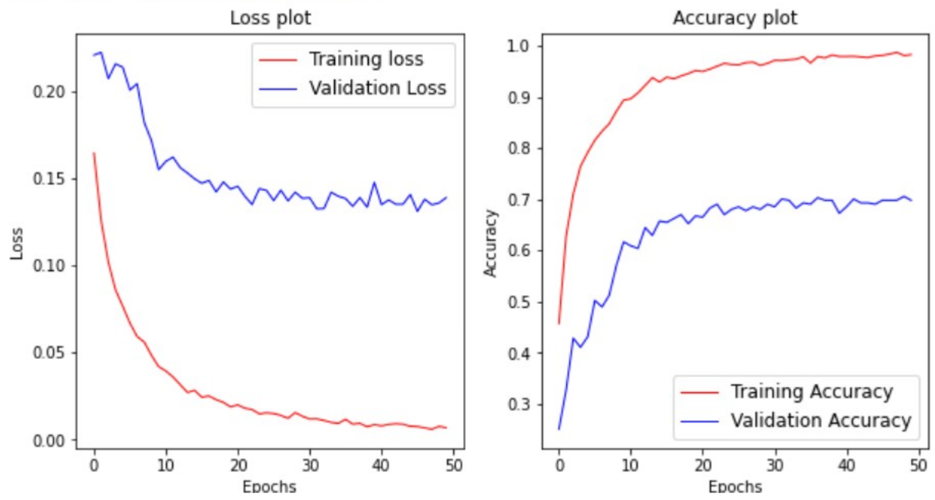


Figure 8. Plots of Loss and Accuracy changes to the training set and test sets for the CNN – MeanSquaredError loss function

The results of this model are pretty much the same as the model before. The accuracy is lower and the validation loss is higher. This method too 24 minutes to train which is almost 21 minutes longer then the previous model with no apparent increase in performance.

V. SUMMARY & KEY FINDINGS

The objective of this experiment was to find the best convolutional neural network that could predict with 99% accuracy the type of tumor present on a brain scan. However the process resulted more difficult then expected and in the end I was not able to archive my objective. I tried many convolutional neural networks by modifying both the layer structure and the hyper-parameters. I also tried to augment the image dataset. However I was not able to find a good solution. The unsolved problem is given by the fact that in all the model where I used '*categorical_crossentropy*' as loss function the validation loss increased as the training went on. As mentioned in section '*IV analyzing the results so far*' this behavior is probably given by the fact that bad predictions are going to be penalized much more than good predictions are rewarded. Which means that, even if many images are correctly predicted, a single misclassification will have a huge loss which will significantly increase the mean loss. That is way, after realizing this, I used the MeanSquaredError loss function to train other model and analyze their results. This choice was fueled by the though that using the MeanSquaredError loss function would help to reduce the penalization for miss-classifications making the model learn better. However, while the validation loss was indeed lower (10 times smaller) the accuracy on the test set could never surpass the 75% mark. I think the main problem here is that no matter what model I choose the image data is too small. The images are all brain scan with brain tumors in them which are all very similar both in size, shape and position of the tumor. Moreover, as mentioned before, not all scans are taken from the same angle making it even more difficult for the algorithm to distinguish between one class to another.

VI. SUGGESTIONS

In this exercise I tried to build many convolutional neural networks with different layer structures and changing all the hyper-parameters in a way I though useful. However I was not able to build a good model. As I explain in the '*Summary & Key Findings*' section I think the best solution is to use much, much bigger dataset, with not just 3 thousand images but with tens of thousands. The reason being that these images are all very similar to one anther with no strong differentiating features. I think a model to be useful need much more training in order to grasp the key features of each class.

Final_course5

March 5, 2021

1 DIARY

2 Machine Learning Final Exercise, Course 5

2.0.1 Background

In this notebook I will work with a collection of images. This image dataset is taken from the Kaggle websites and features 3264 images of brain tumors categorazied in 4 different types. I will use convolutional neural networks to correctly identify the tumor and I will try to inplemet also trasfer learning.

[66]:
```
import numpy as np
import keras
import datetime
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten,␣
 ↪BatchNormalization, LeakyReLU
from keras.layers import Conv2D, MaxPooling2D, AveragePooling2D,␣
 ↪GlobalAveragePooling2D
import matplotlib.pyplot as plt
```

2.0.2 Loading the Training data

[67]:
```
import os, cv2
```

[68]:
```
paths=['glioma_tumor', 'meningioma_tumor', 'no_tumor', 'pituitary_tumor']
```

[69]:
```
x_train = []
y_train = []
#sizes_list=[] #This was to test and verify if the images are all the same size␣
 ↪or not (TEST)

#Looping trough the paths list to load all the images
for i in range(4):

    var_path=('database_c5/Training/%s')%paths[i]
    #Loading every single image from the folder following current path
    for img in os.listdir((var_path)):

        pic = cv2.imread(os.path.join(var_path,img)) #Read the image from the␣
↪folder
        #sizes_list.append(pic.size)                    #(TEST)
        pic = cv2.cvtColor(pic, cv2.COLOR_BGR2GRAY)
        pic = cv2.resize(pic,(200,200))    #this will ensure the same size and␣
↪will downscale them to 200x200.
        pic = np.expand_dims(pic, axis=2) #bigger pictures will use much more␣
↪memory and Im working with my laptop
        x_train.append(pic)
        y_train.append(i)
```

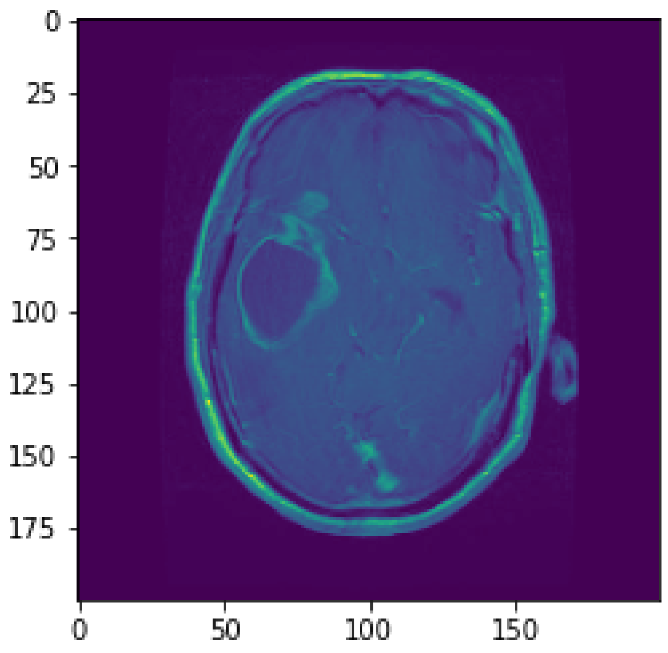
[70]:
```
x_train = np.asarray(x_train)
y_train = np.asarray(y_train)
```

[71]:
```
x_train.shape
```

[71]:
```
(2870, 200, 200, 1)
```

[72]:
```
x_train = x_train.astype('float32')
x_train /= 255
```

[73]:
```
plt.imshow(x_train[23]);
```



```
[74]: y_train[23]
```

```
[74]: 0
```

2.0.3 Loading the Testing data

```
[75]: x_test = []
      y_test = []

      #Looping trough the paths list to load all the images
      for i in range(4):

          var_path=('database_c5/Testing/%s')%paths[i]

          #Loading every single image from the folder following current path
          for img in os.listdir((var_path)):

              pic = cv2.imread(os.path.join(var_path,img)) #Read the image from the␣
       ↪folder
              pic = cv2.cvtColor(pic, cv2.COLOR_BGR2GRAY)
              pic = cv2.resize(pic,(200,200))   #this will ensure the same size and␣
       ↪will downscale them to 200x200.
              pic = np.expand_dims(pic, axis=2) #bigger pictures will use much more␣
       ↪memory and Im working with my laptop
              x_test.append(pic)
              y_test.append(i)
```

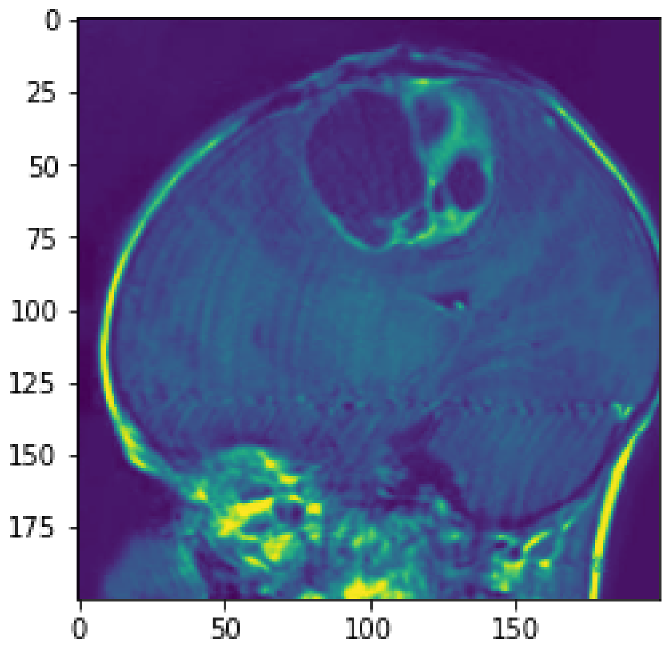
```
[76]: x_test=np.asarray(x_test)
```

```
[77]: x_test.shape
```

```
[77]: (394, 200, 200, 1)
```

```
[78]: x_test = x_test.astype('float32')
      x_test /= 255
```

```
[79]: plt.imshow(x_test[23]);
```



2.0.4 Hot encoding the y_train variable

[80]:
```
classes = 4
y_train = keras.utils.to_categorical(y_train, classes)
y_test = keras.utils.to_categorical(y_test, classes)
```

2.0.5 Getting the timings stright

[81]:
```
time = datetime.datetime.now
time()
```

[81]:
```
datetime.datetime(2021, 3, 5, 14, 45, 51, 101749)
```

2.0.6 CNN implementation

Lets define a function to call when to fit a specific model

[82]:
```
def m_train(model, train, test, number_classes):
    t = time()
    history = model.fit(x_train, y_train,
                        batch_size=16,
                        epochs=15,
                        validation_data=(x_test, y_test),
                        shuffle=True)
    print('Training time: %s' % (time() - t))
    score = model.evaluate(x_test, y_test, verbose=0)
    print('Test score:', score[0])
    print('Test accuracy:', score[1])

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,5))

    ax1.plot(history.history['loss'],'r',linewidth=1.0)
    ax1.plot(history.history['val_loss'],'b',linewidth=1.0)
    ax1.legend(['Training loss', 'Validation Loss'],fontsize=12)
    ax1.set(xlabel='Epochs', ylabel='Loss', title='Loss plot')

    ax2.plot(history.history['accuracy'],'r',linewidth=1.0)
    ax2.plot(history.history['val_accuracy'],'b',linewidth=1.0)
    ax2.legend(['Training Accuracy', 'Validation Accuracy'],fontsize=12)
    ax2.set(xlabel='Epochs', ylabel='Accuracy', title='Accuracy plot')
```

[83]:
```
opt_adam = keras.optimizers.Adam(lr=0.0005, decay=1e-6)
opt = keras.optimizers.RMSprop(lr=0.0005, decay=1e-6)
```

Lets first try the convolutional neural network used in the notebook to see if things actually works.

2.0.7 VANILLA

[90]:
```
#lets first use a 10x10 kernel since my pictures are 200x200,
#with a stride of 2x2 and 32 layers which looked pretty standard
#and with some padding
Cnv_model_layers = [

    Conv2D(32, (10, 10), strides = (5,5), padding='same', input_shape=x_train.
 ↪shape[1:]),
    Activation('relu'),

    Conv2D(32, (10, 10), strides = (5,5)),
    Activation('relu'),

    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),

    Flatten(),

    Dense(512),
    Activation('relu'),
    Dropout(0.5),
    Dense(classes),
    Activation('softmax')
]
```

[91]:
```
Cnv_model = Sequential(Cnv_model_layers)
Cnv_model.summary()
```

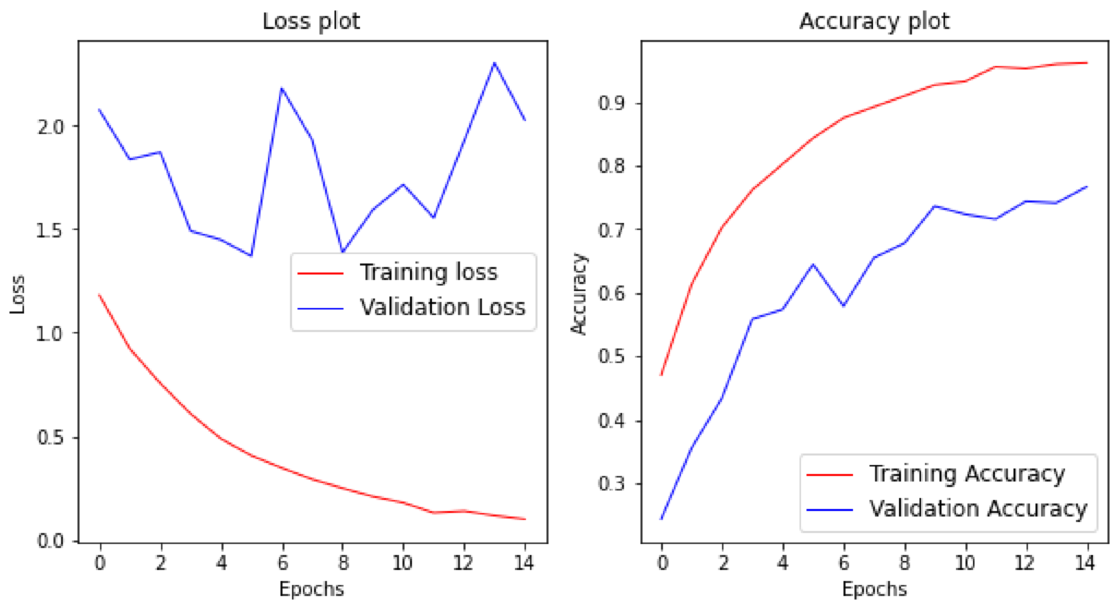
```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_12 (Conv2D)           (None, 40, 40, 32)        3232
_________________________________________________________________
activation_24 (Activation)   (None, 40, 40, 32)        0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 7, 7, 32)          102432
_________________________________________________________________
activation_25 (Activation)   (None, 7, 7, 32)          0
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 3, 3, 32)          0
_________________________________________________________________
dropout_10 (Dropout)         (None, 3, 3, 32)          0
_________________________________________________________________
flatten_6 (Flatten)          (None, 288)               0
_________________________________________________________________
dense_12 (Dense)             (None, 512)               147968
_________________________________________________________________
activation_26 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_11 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_13 (Dense)             (None, 4)                 2052
_________________________________________________________________
activation_27 (Activation)   (None, 4)                 0
=================================================================
Total params: 255,684
Trainable params: 255,684
Non-trainable params: 0
_________________________________________________________________
```

3

I will test this model with just 15 epochs and also the other ones. Just to see the consequences of hyperparameter changes

[140]:
```
Cnv_model.compile(loss='categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])

history = m_train(Cnv_model, x_train, y_train, 4)
```

```
Epoch 1/15
180/180 [==============================] - 5s 27ms/step - loss: 1.2893 -
accuracy: 0.3889 - val_loss: 2.0763 - val_accuracy: 0.2437
Epoch 2/15
180/180 [==============================] - 4s 24ms/step - loss: 0.9849 -
accuracy: 0.5794 - val_loss: 1.8371 - val_accuracy: 0.3553
Epoch 3/15
180/180 [==============================] - 5s 25ms/step - loss: 0.8047 -
accuracy: 0.6758 - val_loss: 1.8725 - val_accuracy: 0.4340
Epoch 4/15
180/180 [==============================] - 5s 26ms/step - loss: 0.6304 -
accuracy: 0.7565 - val_loss: 1.4914 - val_accuracy: 0.5584
Epoch 5/15
180/180 [==============================] - 5s 26ms/step - loss: 0.5008 -
accuracy: 0.8016 - val_loss: 1.4488 - val_accuracy: 0.5736
Epoch 6/15
180/180 [==============================] - 5s 27ms/step - loss: 0.4134 -
accuracy: 0.8375 - val_loss: 1.3700 - val_accuracy: 0.6447
Epoch 7/15
180/180 [==============================] - 5s 26ms/step - loss: 0.3547 -
accuracy: 0.8684 - val_loss: 2.1812 - val_accuracy: 0.5787
Epoch 8/15
180/180 [==============================] - 4s 25ms/step - loss: 0.2825 -
accuracy: 0.8944 - val_loss: 1.9318 - val_accuracy: 0.6548
Epoch 9/15
180/180 [==============================] - 5s 26ms/step - loss: 0.2529 -
accuracy: 0.9073 - val_loss: 1.3872 - val_accuracy: 0.6777
Epoch 10/15
180/180 [==============================] - 5s 26ms/step - loss: 0.2108 -
accuracy: 0.9262 - val_loss: 1.5940 - val_accuracy: 0.7360
Epoch 11/15
180/180 [==============================] - 5s 25ms/step - loss: 0.1719 -
accuracy: 0.9348 - val_loss: 1.7155 - val_accuracy: 0.7234
Epoch 12/15
180/180 [==============================] - 5s 25ms/step - loss: 0.1142 -
accuracy: 0.9613 - val_loss: 1.5554 - val_accuracy: 0.7157
Epoch 13/15
180/180 [==============================] - 5s 25ms/step - loss: 0.1319 -
accuracy: 0.9546 - val_loss: 1.9263 - val_accuracy: 0.7437
Epoch 14/15
180/180 [==============================] - 5s 25ms/step - loss: 0.1129 -
accuracy: 0.9575 - val_loss: 2.3041 - val_accuracy: 0.7411
Epoch 15/15
180/180 [==============================] - 5s 26ms/step - loss: 0.0911 -
accuracy: 0.9643 - val_loss: 2.0265 - val_accuracy: 0.7665
Training time: 0:01:09.979851
Test score: 2.026486873626709
Test accuracy: 0.7664974331855774
```



Ok it works but the validation loss should decrease instead it seems to get higher ! Is it overfitting ? Do i need more data ? Is the regularization dorpout in this case not useful ? Getting a validation loss that increses means that my model is not learning.

Before traing different models lets try to fix this

VANILLA WITHOUT REGULARIZATION

[43]:
```
#Vanilla model without dropout regularization
Cnv_model_noDrop_layers = [

    Conv2D(32, (10, 10), strides = (5,5), padding='same', input_shape=x_train.
 ↪shape[1:]),
    Activation('relu'),

    Conv2D(32, (10, 10), strides = (5,5)),
    Activation('relu'),

    MaxPooling2D(pool_size=(2, 2)),
    #Dropout(0.25),

    Flatten(),

    Dense(512),
    Activation('relu'),
    #Dropout(0.5),
    Dense(classes),
    Activation('softmax')
]
```

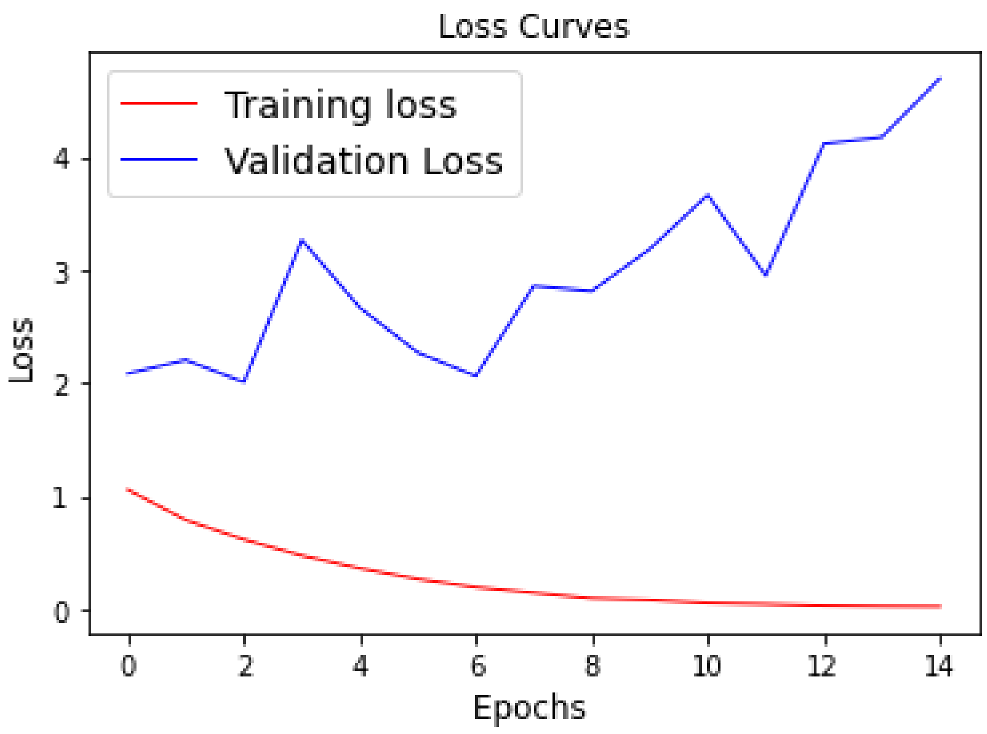
[44]:
```
Cnv_model_noDrop = Sequential(Cnv_model_noDrop_layers)
Cnv_model_noDrop.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_6 (Conv2D)            (None, 40, 40, 32)        3232
_________________________________________________________________
activation_12 (Activation)   (None, 40, 40, 32)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 7, 7, 32)          102432
_________________________________________________________________
activation_13 (Activation)   (None, 7, 7, 32)          0
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 3, 3, 32)          0
_________________________________________________________________
flatten_3 (Flatten)          (None, 288)               0
_________________________________________________________________
dense_6 (Dense)              (None, 512)               147968
_________________________________________________________________
activation_14 (Activation)   (None, 512)               0
_________________________________________________________________
dense_7 (Dense)              (None, 4)                 2052
_________________________________________________________________
activation_15 (Activation)   (None, 4)                 0
=================================================================
Total params: 255,684
Trainable params: 255,684
Non-trainable params: 0
_________________________________________________________________
```

[39]:
```
Cnv_model_noDrop.compile(loss='categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])

m_train(Cnv_model_noDrop, x_train, y_train, 4)
```

```
Epoch 1/15
180/180 [==============================] - 5s 27ms/step - loss: 1.1906 -
accuracy: 0.4599 - val_loss: 2.0841 - val_accuracy: 0.2817
Epoch 2/15
180/180 [==============================] - 5s 26ms/step - loss: 0.8200 -
accuracy: 0.6581 - val_loss: 2.2039 - val_accuracy: 0.3756
Epoch 3/15
```

```
180/180 [==============================] - 5s 26ms/step - loss: 0.6467 -
accuracy: 0.7349 - val_loss: 2.0086 - val_accuracy: 0.5279
Epoch 4/15
180/180 [==============================] - 5s 25ms/step - loss: 0.5041 -
accuracy: 0.7956 - val_loss: 3.2647 - val_accuracy: 0.4619
Epoch 5/15
180/180 [==============================] - 5s 26ms/step - loss: 0.3573 -
accuracy: 0.8639 - val_loss: 2.6669 - val_accuracy: 0.6066
Epoch 6/15
180/180 [==============================] - 5s 26ms/step - loss: 0.2651 -
accuracy: 0.8988 - val_loss: 2.2699 - val_accuracy: 0.6701
Epoch 7/15
180/180 [==============================] - 5s 25ms/step - loss: 0.1829 -
accuracy: 0.9335 - val_loss: 2.0624 - val_accuracy: 0.7208
Epoch 8/15
180/180 [==============================] - 4s 24ms/step - loss: 0.1390 -
accuracy: 0.9541 - val_loss: 2.8600 - val_accuracy: 0.6980
Epoch 9/15
180/180 [==============================] - 5s 26ms/step - loss: 0.1020 -
accuracy: 0.9670 - val_loss: 2.8134 - val_accuracy: 0.7234
Epoch 10/15
180/180 [==============================] - 5s 25ms/step - loss: 0.0836 -
accuracy: 0.9741 - val_loss: 3.1844 - val_accuracy: 0.7132
Epoch 11/15
180/180 [==============================] - 5s 26ms/step - loss: 0.0567 -
accuracy: 0.9867 - val_loss: 3.6632 - val_accuracy: 0.7234
Epoch 12/15
180/180 [==============================] - 5s 28ms/step - loss: 0.0353 -
accuracy: 0.9909 - val_loss: 2.9495 - val_accuracy: 0.7360
Epoch 13/15
180/180 [==============================] - 5s 25ms/step - loss: 0.0380 -
accuracy: 0.9894 - val_loss: 4.1158 - val_accuracy: 0.7487
Epoch 14/15
180/180 [==============================] - 5s 26ms/step - loss: 0.0279 -
accuracy: 0.9931 - val_loss: 4.1744 - val_accuracy: 0.7766
Epoch 15/15
180/180 [==============================] - 5s 26ms/step - loss: 0.0228 -
accuracy: 0.9922 - val_loss: 4.6907 - val_accuracy: 0.7538
Training time: 0:01:10.327924
Test score: 4.690695285797119
Test accuracy: 0.7538071274757385
```



Not much of a difference. I think I will try to use leaky relu or tanh as activation functions

VANILLA WITH LEAKY RELU ACTIVATION FUNCTION

[141]:
```
#With dropout but with different activation function - leaky relu
Cnv_model_leaky_layers = [

    Conv2D(32, (10, 10), strides = (5,5), padding='same', input_shape=x_train.
 ↪shape[1:]),
    LeakyReLU(alpha=0.3),

    Conv2D(32, (10, 10), strides = (5,5)),
    LeakyReLU(alpha=0.3),

    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),

    Flatten(),

    Dense(512),
    LeakyReLU(alpha=0.3),
    Dropout(0.5),
    Dense(classes),
    Activation('softmax')
]
```

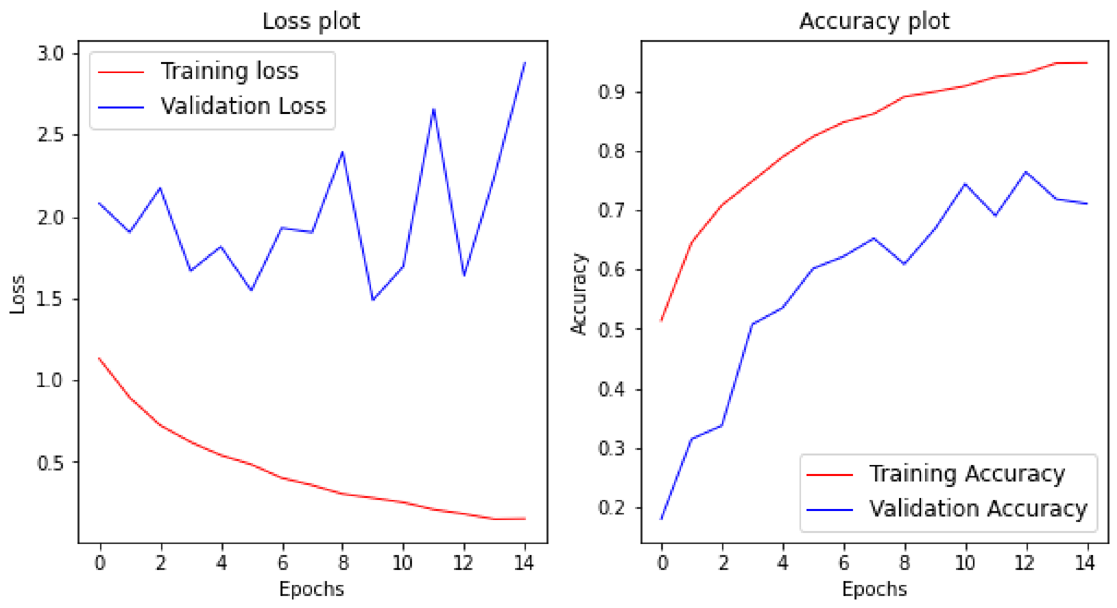
[142]:
```
Cnv_model_leaky = Sequential(Cnv_model_leaky_layers)
Cnv_model_leaky.summary()
```

```
Model: "sequential_22"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_40 (Conv2D)           (None, 40, 40, 32)        3232
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 40, 40, 32)        0
_________________________________________________________________
conv2d_41 (Conv2D)           (None, 7, 7, 32)          102432
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 7, 7, 32)          0
_________________________________________________________________
max_pooling2d_20 (MaxPooling (None, 3, 3, 32)          0
_________________________________________________________________
dropout_40 (Dropout)         (None, 3, 3, 32)          0
_________________________________________________________________
flatten_20 (Flatten)         (None, 288)               0
_________________________________________________________________
dense_40 (Dense)             (None, 512)               147968
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 512)               0
_________________________________________________________________
dropout_41 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_41 (Dense)             (None, 4)                 2052
_________________________________________________________________
activation_80 (Activation)   (None, 4)                 0
=================================================================
Total params: 255,684
Trainable params: 255,684
Non-trainable params: 0

_________________________________________________________________
```

[143]:
```
Cnv_model_leaky.compile(loss='categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])

m_train(Cnv_model_leaky, x_train, y_train, 4)
```

```
Epoch 1/15
180/180 [==============================] - 5s 26ms/step - loss: 1.2432 -
accuracy: 0.4340 - val_loss: 2.0791 - val_accuracy: 0.1802
```

```
Epoch 2/15
180/180 [==============================] - 4s 25ms/step - loss: 0.9251 -
accuracy: 0.6354 - val_loss: 1.9026 - val_accuracy: 0.3147
Epoch 3/15
180/180 [==============================] - 5s 26ms/step - loss: 0.7299 -
accuracy: 0.7055 - val_loss: 2.1728 - val_accuracy: 0.3376
Epoch 4/15
180/180 [==============================] - 5s 28ms/step - loss: 0.6276 -
accuracy: 0.7489 - val_loss: 1.6671 - val_accuracy: 0.5076
Epoch 5/15
180/180 [==============================] - 5s 28ms/step - loss: 0.6015 -
accuracy: 0.7580 - val_loss: 1.8154 - val_accuracy: 0.5355
Epoch 6/15
180/180 [==============================] - 5s 25ms/step - loss: 0.4478 -
accuracy: 0.8341 - val_loss: 1.5462 - val_accuracy: 0.6015
Epoch 7/15
180/180 [==============================] - 4s 25ms/step - loss: 0.3924 -
accuracy: 0.8537 - val_loss: 1.9289 - val_accuracy: 0.6218
Epoch 8/15
180/180 [==============================] - 5s 25ms/step - loss: 0.3508 -
accuracy: 0.8619 - val_loss: 1.9023 - val_accuracy: 0.6523
Epoch 9/15
180/180 [==============================] - 5s 26ms/step - loss: 0.3139 -
accuracy: 0.8843 - val_loss: 2.3940 - val_accuracy: 0.6091
Epoch 10/15
180/180 [==============================] - 5s 26ms/step - loss: 0.2558 -
accuracy: 0.9028 - val_loss: 1.4882 - val_accuracy: 0.6675
Epoch 11/15
180/180 [==============================] - 4s 25ms/step - loss: 0.2311 -
accuracy: 0.9138 - val_loss: 1.6939 - val_accuracy: 0.7437
Epoch 12/15
180/180 [==============================] - 4s 25ms/step - loss: 0.1973 -
accuracy: 0.9299 - val_loss: 2.6572 - val_accuracy: 0.6904
Epoch 13/15
180/180 [==============================] - 4s 25ms/step - loss: 0.1659 -
accuracy: 0.9375 - val_loss: 1.6395 - val_accuracy: 0.7640
Epoch 14/15
180/180 [==============================] - 4s 25ms/step - loss: 0.1475 -
accuracy: 0.9481 - val_loss: 2.2451 - val_accuracy: 0.7183
Epoch 15/15
180/180 [==============================] - 5s 26ms/step - loss: 0.1470 -
accuracy: 0.9467 - val_loss: 2.9382 - val_accuracy: 0.7107
Training time: 0:01:09.838030
Test score: 2.9382076263427734
Test accuracy: 0.710659921169281
```



VANILLA WITH TANH ACTIVATION FUNCTION

[144]:
```
Cnv_model_tanh_layers = [

    Conv2D(32, (10, 10), strides = (5,5), padding='same', input_shape=x_train.
 ↪shape[1:]),
    Activation('tanh'),

    Conv2D(32, (10, 10), strides = (5,5)),
    Activation('tanh'),

    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),

    Flatten(),

    Dense(512),
    Activation('tanh'),
    Dropout(0.5),
    Dense(classes),
    Activation('softmax')
]
```

[145]:
```
Cnv_model_tanh = Sequential(Cnv_model_tanh_layers)
Cnv_model_tanh.summary()
```

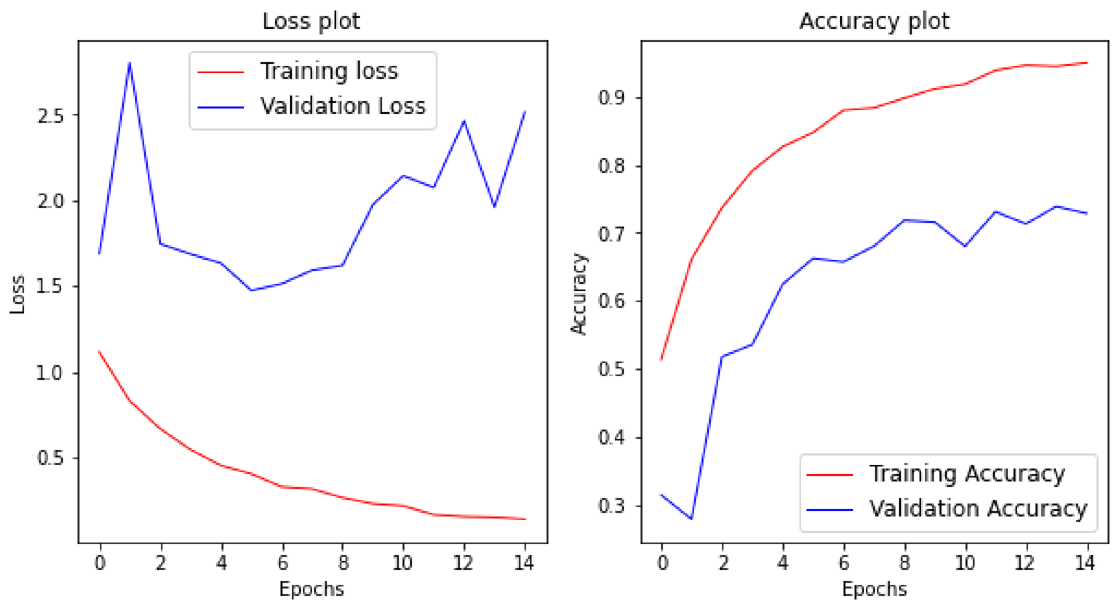
```
Model: "sequential_23"
_________________________________________________________________
```

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_42 (Conv2D)           (None, 40, 40, 32)        3232
_________________________________________________________________
activation_81 (Activation)   (None, 40, 40, 32)        0
_________________________________________________________________
conv2d_43 (Conv2D)           (None, 7, 7, 32)          102432
_________________________________________________________________
activation_82 (Activation)   (None, 7, 7, 32)          0
_________________________________________________________________
max_pooling2d_21 (MaxPooling (None, 3, 3, 32)          0
_________________________________________________________________
dropout_42 (Dropout)         (None, 3, 3, 32)          0
_________________________________________________________________
flatten_21 (Flatten)         (None, 288)               0
_________________________________________________________________
dense_42 (Dense)             (None, 512)               147968
_________________________________________________________________
activation_83 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_43 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_43 (Dense)             (None, 4)                 2052
_________________________________________________________________
activation_84 (Activation)   (None, 4)                 0
=================================================================
Total params: 255,684
Trainable params: 255,684
Non-trainable params: 0
_________________________________________________________________
```

[146]:
```
Cnv_model_tanh.compile(loss='categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])

m_train(Cnv_model_tanh, x_train, y_train, 4)
```

```
Epoch 1/15
180/180 [==============================] - 5s 25ms/step - loss: 1.2338 -
accuracy: 0.4449 - val_loss: 1.6884 - val_accuracy: 0.3147
Epoch 2/15
180/180 [==============================] - 4s 25ms/step - loss: 0.8877 -
accuracy: 0.6486 - val_loss: 2.8017 - val_accuracy: 0.2792
Epoch 3/15
180/180 [==============================] - 4s 24ms/step - loss: 0.6865 -
accuracy: 0.7286 - val_loss: 1.7453 - val_accuracy: 0.5178
Epoch 4/15
180/180 [==============================] - 5s 25ms/step - loss: 0.5356 -
```

```
accuracy: 0.8035 - val_loss: 1.6854 - val_accuracy: 0.5355
Epoch 5/15
180/180 [==============================] - 4s 24ms/step - loss: 0.4594 -
accuracy: 0.8357 - val_loss: 1.6332 - val_accuracy: 0.6244
Epoch 6/15
180/180 [==============================] - 5s 27ms/step - loss: 0.3771 -
accuracy: 0.8527 - val_loss: 1.4731 - val_accuracy: 0.6624
Epoch 7/15
180/180 [==============================] - 5s 26ms/step - loss: 0.3369 -
accuracy: 0.8846 - val_loss: 1.5126 - val_accuracy: 0.6574
Epoch 8/15
180/180 [==============================] - 5s 27ms/step - loss: 0.3190 -
accuracy: 0.8847 - val_loss: 1.5914 - val_accuracy: 0.6802
Epoch 9/15
180/180 [==============================] - 5s 27ms/step - loss: 0.2461 -
accuracy: 0.9033 - val_loss: 1.6199 - val_accuracy: 0.7183
Epoch 10/15
180/180 [==============================] - 5s 28ms/step - loss: 0.2173 -
accuracy: 0.9158 - val_loss: 1.9742 - val_accuracy: 0.7157
Epoch 11/15
180/180 [==============================] - 5s 26ms/step - loss: 0.2041 -
accuracy: 0.9245 - val_loss: 2.1425 - val_accuracy: 0.6802
Epoch 12/15
180/180 [==============================] - 5s 25ms/step - loss: 0.1778 -
accuracy: 0.9374 - val_loss: 2.0740 - val_accuracy: 0.7310
Epoch 13/15
180/180 [==============================] - 5s 26ms/step - loss: 0.1422 -
accuracy: 0.9510 - val_loss: 2.4624 - val_accuracy: 0.7132
Epoch 14/15
180/180 [==============================] - 5s 26ms/step - loss: 0.1400 -
accuracy: 0.9499 - val_loss: 1.9586 - val_accuracy: 0.7386
Epoch 15/15
180/180 [==============================] - 5s 26ms/step - loss: 0.1344 -
accuracy: 0.9510 - val_loss: 2.5146 - val_accuracy: 0.7284
Training time: 0:01:10.565618
Test score: 2.514586925506592
Test accuracy: 0.7284263968467712
```



I cant figure out why my model refuse to learn. maybe I dont have enough images to train on. Changing the activation function doenst seems to have solved the problem so I will stick with the relu activation function.

Actually It just came to my mind that maybe the batch size I choose is too small ! I thought that since did have a huge amount of images that it was better to have many batches, but maybe im completely wrong and if the batch sizes ar too small maybe the model can train well. With a smaller batch size the model will train on more batches but these batches are probably not significative.

Lets try to increase the batch size !

I tried batch size equal to 8, 16, 32, 64, 128, 256. As the batch sizes increses the validatin loss get worse and worse. the best ones where batch size 16 and 32

[20]:
```
def m_train32(model, train, test, number_classes):
    t = time()
    history = model.fit(x_train, y_train,
                        batch_size=32,
                        epochs=15,
                        validation_data=(x_test, y_test),
                        shuffle=True)

    print('Training time: %s' % (time() - t))
    score = model.evaluate(x_test, y_test, verbose=0)
    print('Test score:', score[0])
    print('Test accuracy:', score[1])

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,5))
    ax1.plot(history.history['loss'],'r',linewidth=1.0)
    ax1.plot(history.history['val_loss'],'b',linewidth=1.0)
    ax1.legend(['Training loss', 'Validation Loss'],fontsize=12)
    ax1.set(xlabel='Epochs', ylabel='Loss', title='Loss plot')

    ax2.plot(history.history['accuracy'],'r',linewidth=1.0)
    ax2.plot(history.history['val_accuracy'],'b',linewidth=1.0)
    ax2.legend(['Training Accuracy', 'Validation Accuracy'],fontsize=12)
    ax2.set(xlabel='Epochs', ylabel='Accuracy', title='Accuracy plot')
```

I will try to change the filter depth to see if there is an improvement and i will change the kernel and stride. I know that a 5x5 stide is huge but it was my first choice in order to run multiple fast networks. Now lets see if its the stride that is too big and keeps the models from learning

3.0.1 Lets see if a more complex network will do the job

Lets first spice it up with using a depth of 128

3.0.2 CONV version 1 - Filters 128

[148]:
```
Cnv_model_v1_layers = [
    Conv2D(128, (10, 10), strides = (5,5), padding='same', input_shape=x_train.
 ↪shape[1:]),
    Activation('relu'),

    Conv2D(128, (10, 10), strides = (5,5)),
    Activation('relu'),

    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),

    Flatten(),

    Dense(512),
    Activation('relu'),
    Dropout(0.5),
    Dense(classes),
    Activation('softmax')
]
```

[149]:
```
Cnv_model_v1 = Sequential(Cnv_model_v1_layers)
Cnv_model_v1.summary()
```

```
Model: "sequential_24"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_44 (Conv2D)           (None, 40, 40, 128)       12928
```

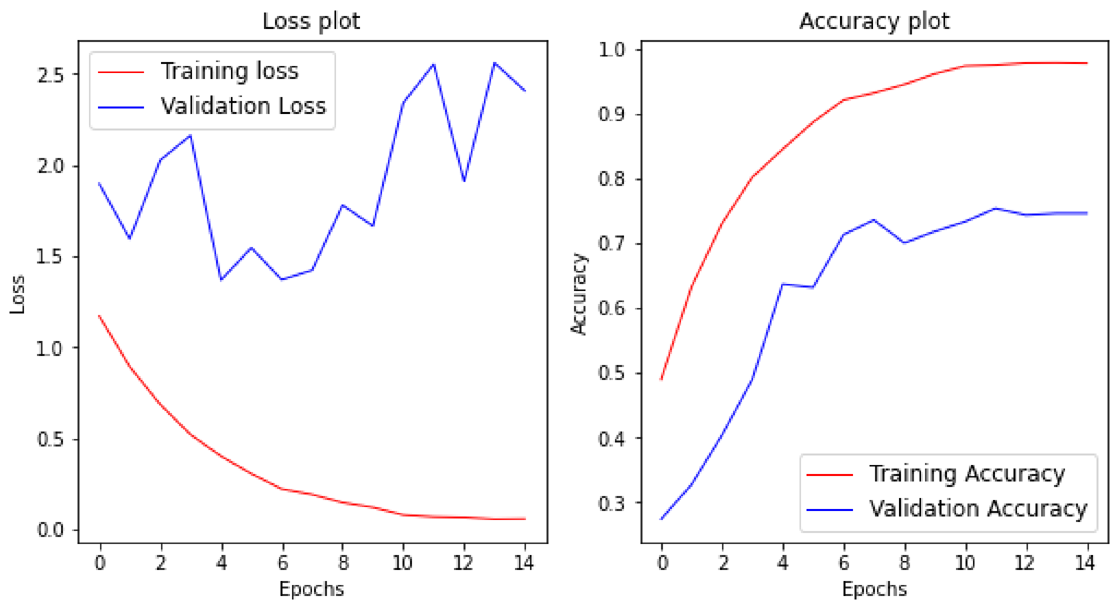
```
_________________________________________________________________
activation_85 (Activation)   (None, 40, 40, 128)       0
_________________________________________________________________
conv2d_45 (Conv2D)           (None, 7, 7, 128)         1638528
_________________________________________________________________
activation_86 (Activation)   (None, 7, 7, 128)         0
_________________________________________________________________
max_pooling2d_22 (MaxPooling (None, 3, 3, 128)         0
_________________________________________________________________
dropout_44 (Dropout)         (None, 3, 3, 128)         0
_________________________________________________________________
flatten_22 (Flatten)         (None, 1152)              0
_________________________________________________________________
dense_44 (Dense)             (None, 512)               590336
_________________________________________________________________
activation_87 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_45 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_45 (Dense)             (None, 4)                 2052
_________________________________________________________________
activation_88 (Activation)   (None, 4)                 0
=================================================================
Total params: 2,243,844
Trainable params: 2,243,844
Non-trainable params: 0
_________________________________________________________________
```

ten times more params !

[150]:
```
Cnv_model_v1.compile(loss='categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])

m_train32(Cnv_model_v1, x_train, y_train, 4)
```

```
Epoch 1/15
90/90 [==============================] - 26s 288ms/step - loss: 1.2838 -
accuracy: 0.4241 - val_loss: 1.8964 - val_accuracy: 0.2741
Epoch 2/15
90/90 [==============================] - 26s 286ms/step - loss: 0.9344 -
accuracy: 0.6168 - val_loss: 1.5942 - val_accuracy: 0.3274
Epoch 3/15
90/90 [==============================] - 25s 283ms/step - loss: 0.6973 -
accuracy: 0.7203 - val_loss: 2.0264 - val_accuracy: 0.4036
Epoch 4/15
90/90 [==============================] - 25s 283ms/step - loss: 0.5450 -
accuracy: 0.7914 - val_loss: 2.1625 - val_accuracy: 0.4898
```

```
Epoch 5/15
90/90 [==============================] - 25s 281ms/step - loss: 0.3885 -
accuracy: 0.8491 - val_loss: 1.3677 - val_accuracy: 0.6371
Epoch 6/15
90/90 [==============================] - 26s 285ms/step - loss: 0.3103 -
accuracy: 0.8855 - val_loss: 1.5458 - val_accuracy: 0.6320
Epoch 7/15
90/90 [==============================] - 25s 283ms/step - loss: 0.2156 -
accuracy: 0.9215 - val_loss: 1.3703 - val_accuracy: 0.7132
Epoch 8/15
90/90 [==============================] - 26s 284ms/step - loss: 0.2074 -
accuracy: 0.9246 - val_loss: 1.4225 - val_accuracy: 0.7360
Epoch 9/15
90/90 [==============================] - 26s 285ms/step - loss: 0.1353 -
accuracy: 0.9537 - val_loss: 1.7799 - val_accuracy: 0.7005
Epoch 10/15
90/90 [==============================] - 26s 285ms/step - loss: 0.1062 -
accuracy: 0.9625 - val_loss: 1.6654 - val_accuracy: 0.7183
Epoch 11/15
90/90 [==============================] - 25s 283ms/step - loss: 0.0666 -
accuracy: 0.9800 - val_loss: 2.3413 - val_accuracy: 0.7335
Epoch 12/15
90/90 [==============================] - 29s 328ms/step - loss: 0.0661 -
accuracy: 0.9749 - val_loss: 2.5531 - val_accuracy: 0.7538
Epoch 13/15
90/90 [==============================] - 29s 320ms/step - loss: 0.0485 -
accuracy: 0.9833 - val_loss: 1.9106 - val_accuracy: 0.7437
Epoch 14/15
90/90 [==============================] - 28s 310ms/step - loss: 0.0548 -
accuracy: 0.9798 - val_loss: 2.5614 - val_accuracy: 0.7462
Epoch 15/15
90/90 [==============================] - 29s 317ms/step - loss: 0.0520 -
accuracy: 0.9797 - val_loss: 2.4069 - val_accuracy: 0.7462
Training time: 0:06:36.578635
Test score: 2.4069085121154785
Test accuracy: 0.7461928725242615
```



The filter detph seems to give a lower overall validation loss it is still not dropping as it should

Next step is to to use a smaller kernel of 3x3 and a stride of 2x2 in the first CNN I did.

3.0.3 CONV version2

[151]:
```
#lets use a smaller kernel
#with a stride of 2x2 and 32 layers which looked pretty standard
#and with some padding

Cnv_model_v2_layers = [

    Conv2D(32, (3, 3), strides = (1,1), padding='same', input_shape=x_train.
 ↪shape[1:]),
    Activation('relu'),
    Conv2D(32, (3, 3), strides = (1,1)),
    Activation('relu'),

    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),

    Conv2D(32, (3, 3), strides = (1,1), padding='same', input_shape=x_train.
 ↪shape[1:]),
    Activation('relu'),
    Conv2D(32, (3, 3), strides = (1,1)),
    Activation('relu'),

    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),

    Conv2D(32, (3, 3), strides = (1,1), padding='same', input_shape=x_train.
 ↪shape[1:]),
    Activation('relu'),
    Conv2D(32, (3, 3), strides = (1,1)),
    Activation('relu'),

    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),

    Flatten(),

    Dense(512),
    Activation('relu'),
    Dropout(0.5),
    Dense(classes),
    Activation('softmax')
]
```

[152]:
```
Cnv_model_v2 = Sequential(Cnv_model_v2_layers)
Cnv_model_v2.summary()
```

```
Model: "sequential_25"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_46 (Conv2D)           (None, 200, 200, 32)      320
_________________________________________________________________
activation_89 (Activation)   (None, 200, 200, 32)      0
_________________________________________________________________
conv2d_47 (Conv2D)           (None, 198, 198, 32)      9248
_________________________________________________________________
activation_90 (Activation)   (None, 198, 198, 32)      0
_________________________________________________________________
max_pooling2d_23 (MaxPooling (None, 99, 99, 32)        0
_________________________________________________________________
dropout_46 (Dropout)         (None, 99, 99, 32)        0
_________________________________________________________________
conv2d_48 (Conv2D)           (None, 99, 99, 32)        9248
_________________________________________________________________
activation_91 (Activation)   (None, 99, 99, 32)        0
_________________________________________________________________
conv2d_49 (Conv2D)           (None, 97, 97, 32)        9248
_________________________________________________________________
activation_92 (Activation)   (None, 97, 97, 32)        0
_________________________________________________________________
```

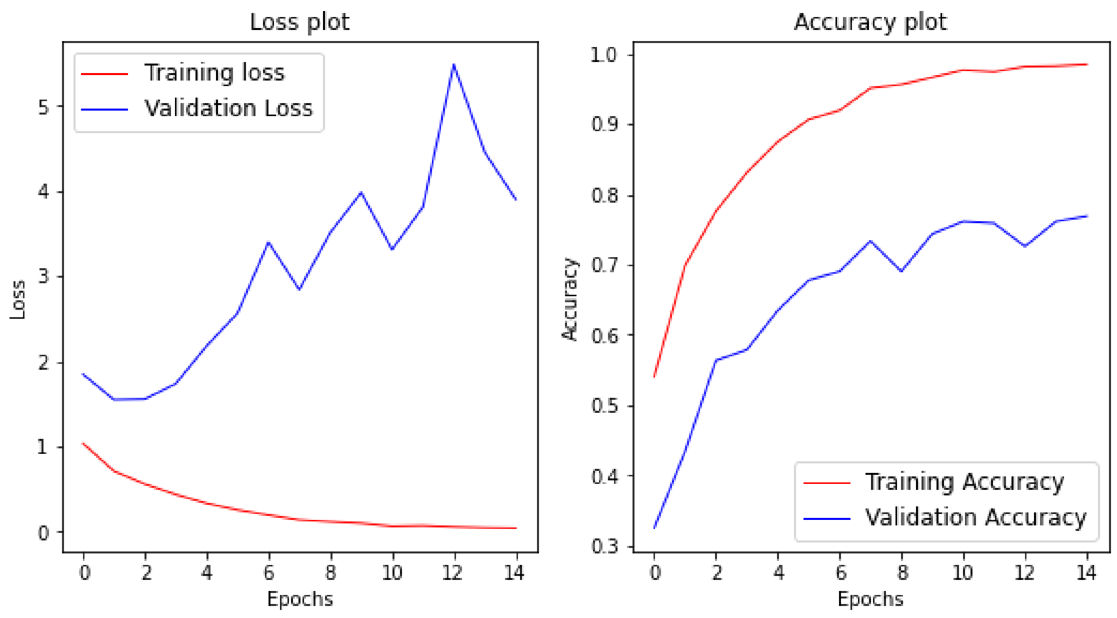
```
max_pooling2d_24 (MaxPooling (None, 48, 48, 32)        0

_________________________________________________________________
dropout_47 (Dropout)         (None, 48, 48, 32)        0

_________________________________________________________________
conv2d_50 (Conv2D)           (None, 48, 48, 32)        9248

_________________________________________________________________
activation_93 (Activation)   (None, 48, 48, 32)        0

_________________________________________________________________
conv2d_51 (Conv2D)           (None, 46, 46, 32)        9248

_________________________________________________________________
activation_94 (Activation)   (None, 46, 46, 32)        0

_________________________________________________________________
max_pooling2d_25 (MaxPooling (None, 23, 23, 32)        0

_________________________________________________________________
dropout_48 (Dropout)         (None, 23, 23, 32)        0

_________________________________________________________________
flatten_23 (Flatten)         (None, 16928)             0

_________________________________________________________________
dense_46 (Dense)             (None, 512)               8667648

_________________________________________________________________
activation_95 (Activation)   (None, 512)               0

_________________________________________________________________
dropout_49 (Dropout)         (None, 512)               0

_________________________________________________________________
dense_47 (Dense)             (None, 4)                 2052

_________________________________________________________________
activation_96 (Activation)   (None, 4)                 0
=================================================================
Total params: 8,716,260
Trainable params: 8,716,260
Non-trainable params: 0

_________________________________________________________________
```

More then the kernel size is the stride size that increases the total parameters to be calculated.

[153]:
```
Cnv_model_v2.compile(loss='categorical_crossentropy', #we always need␣
↪categorical_crossentropy, i think.
              optimizer=opt_adam,
              metrics=['accuracy'])

m_train32(Cnv_model_v2, x_train, y_train, 4)
```

```
Epoch 1/15
90/90 [==============================] - 243s 3s/step - loss: 1.1758 - accuracy:
0.4723 - val_loss: 1.8493 - val_accuracy: 0.3249
Epoch 2/15
90/90 [==============================] - 243s 3s/step - loss: 0.7371 - accuracy:
0.6927 - val_loss: 1.5501 - val_accuracy: 0.4340
```

```
Epoch 3/15
90/90 [==============================] - 243s 3s/step - loss: 0.5868 - accuracy:
0.7616 - val_loss: 1.5574 - val_accuracy: 0.5635
Epoch 4/15
90/90 [==============================] - 241s 3s/step - loss: 0.4313 - accuracy:
0.8284 - val_loss: 1.7373 - val_accuracy: 0.5787
Epoch 5/15
90/90 [==============================] - 237s 3s/step - loss: 0.3486 - accuracy:
0.8669 - val_loss: 2.1815 - val_accuracy: 0.6345
Epoch 6/15
90/90 [==============================] - 240s 3s/step - loss: 0.2495 - accuracy:
0.9055 - val_loss: 2.5649 - val_accuracy: 0.6777
Epoch 7/15
90/90 [==============================] - 236s 3s/step - loss: 0.2083 - accuracy:
0.9155 - val_loss: 3.3998 - val_accuracy: 0.6904
Epoch 8/15
90/90 [==============================] - 236s 3s/step - loss: 0.1392 - accuracy:
0.9524 - val_loss: 2.8414 - val_accuracy: 0.7335
Epoch 9/15
90/90 [==============================] - 236s 3s/step - loss: 0.1148 - accuracy:
0.9552 - val_loss: 3.5123 - val_accuracy: 0.6904
Epoch 10/15
90/90 [==============================] - 236s 3s/step - loss: 0.1018 - accuracy:
0.9646 - val_loss: 3.9814 - val_accuracy: 0.7437
Epoch 11/15
90/90 [==============================] - 246s 3s/step - loss: 0.0603 - accuracy:
0.9799 - val_loss: 3.3125 - val_accuracy: 0.7614
Epoch 12/15
90/90 [==============================] - 251s 3s/step - loss: 0.0656 - accuracy:
0.9767 - val_loss: 3.8126 - val_accuracy: 0.7589
Epoch 13/15
90/90 [==============================] - 241s 3s/step - loss: 0.0684 - accuracy:
0.9787 - val_loss: 5.4869 - val_accuracy: 0.7259
Epoch 14/15
90/90 [==============================] - 237s 3s/step - loss: 0.0528 - accuracy:
0.9828 - val_loss: 4.4535 - val_accuracy: 0.7614
Epoch 15/15
90/90 [==============================] - 236s 3s/step - loss: 0.0424 - accuracy:
0.9851 - val_loss: 3.9023 - val_accuracy: 0.7690
Training time: 1:00:03.675333
Test score: 3.9023020267486572
Test accuracy: 0.7690355181694031
```



It took 1 hour to run and there is still the same issue with the validation loss.

3.0.4 CONV Version3 - EXPLODED

Ok, maybe its better not to run this one. Dense layers are the main source of parameters which makes the idea of adding many dense layers not very apealing for someone like me that is using his laptop

Lets try another one with adding more conv layers and max pooling layers.

interesting, very few parameters

3.0.5 CONV Version 3 - v2 with Less layers

[21]:
```
Cnv_model_v3_layers = [
    Conv2D(32, (3, 3), padding='same', input_shape=x_train.shape[1:]),
    Activation('relu'),

    Conv2D(32, (3, 3)),
    Activation('relu'),

    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),

    Flatten(),
    Dense(512),
    Activation('relu'),
    Dropout(0.5),
    Dense(classes),
    Activation('softmax')
]
```

[22]:
```
Cnv_model_v3 = Sequential(Cnv_model_v3_layers)
Cnv_model_v3.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 200, 200, 32)      320
_________________________________________________________________
activation (Activation)      (None, 200, 200, 32)      0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 198, 198, 32)      9248
_________________________________________________________________
activation_1 (Activation)    (None, 198, 198, 32)      0
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 99, 99, 32)        0
_________________________________________________________________
dropout (Dropout)            (None, 99, 99, 32)        0
_________________________________________________________________
flatten (Flatten)            (None, 313632)            0
_________________________________________________________________
dense (Dense)                (None, 512)               160580096
_________________________________________________________________
activation_2 (Activation)    (None, 512)               0
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 4)                 2052
_________________________________________________________________
activation_3 (Activation)    (None, 4)                 0
=================================================================
Total params: 160,591,716
Trainable params: 160,591,716
Non-trainable params: 0
_________________________________________________________________
```

[ ]:
```
Cnv_model_v3.compile(loss='categorical_crossentropy', #we always need␣
 ↪categorical_crossentropy, i think.
              optimizer=opt_adam,
              metrics=['accuracy'])

m_train32(Cnv_model_v3, x_train, y_train, 4)
```

```
Epoch 1/15
 3/90 [>…] - ETA: 4:11 - loss: 3.4037 - accuracy:
0.2969
```

ah ! now i see. A deeper network allows me to use a smaller stride without a huge increase of the parameters.

Now i would like to take the Conv_model_v2 and add more dense layers

3.0.6 CONV version 2 - more dense layers

```
[ ]: Cnv_model_v2_2_layers = [
         Conv2D(32, (3, 3), strides = (1,1), padding='same', input_shape=x_train.
     ↪shape[1:]),
         Activation('relu'),
         Conv2D(32, (3, 3), strides = (1,1)),
         Activation('relu'),

         MaxPooling2D(pool_size=(2, 2)),
         Dropout(0.25),

         Conv2D(32, (3, 3), strides = (1,1), padding='same', input_shape=x_train.
     ↪shape[1:]),
         Activation('relu'),
         Conv2D(32, (3, 3), strides = (1,1)),
         Activation('relu'),

         MaxPooling2D(pool_size=(2, 2)),
         Dropout(0.25),

         Flatten(),

         Dense(512),
         Activation('relu'),
         Dense(256),
         Activation('relu'),
         Dense(128),
         Activation('relu'),
         Dropout(0.5),
         Dense(classes),
         Activation('softmax')
     ]
```

```
[ ]: Cnv_model_v2_2 = Sequential(Cnv_model_v2_2_layers)
     Cnv_model_v2_2.summary()
```

```
[ ]: Cnv_model_v2_2.compile(loss='categorical_crossentropy',
                   optimizer=opt_adam,
            metrics=['accuracy'])

m_train(Cnv_model_v2_2, x_train, y_train, 4)
```

```
Epoch 1/15
  1/180 […] - ETA: 6:16 - loss: 1.3923 - accuracy:
0.2500
```

Increasing the number of dense layers doesn't really improve our model but its reather worsening it.

3.0.7 At this point, I will use my last resource - DATA AUGMENTATION

[38]:
```
def m_train_aug(model, train, test, number_classes):
    t = time()
    history = model.fit(datagen.flow(x_train, y_train, batch_size=32),␣
 ↪#generating 32 random images each epoc
                       steps_per_epoch=len(x_train) / 32,
                       epochs=25,
                       validation_data=(x_test, y_test),
                       shuffle=True)

    print('Training time: %s' % (time() - t))
    score = model.evaluate(x_test, y_test, verbose=0)
    print('Test score:', score[0])
    print('Test accuracy:', score[1])

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,5))

    ax1.plot(history.history['loss'],'r',linewidth=1.0)
    ax1.plot(history.history['val_loss'],'b',linewidth=1.0)
    ax1.legend(['Training loss', 'Validation Loss'],fontsize=12)
    ax1.set(xlabel='Epochs', ylabel='Loss', title='Loss plot')

    ax2.plot(history.history['accuracy'],'r',linewidth=1.0)
    ax2.plot(history.history['val_accuracy'],'b',linewidth=1.0)
    ax2.legend(['Training Accuracy', 'Validation Accuracy'],fontsize=12)
    ax2.set(xlabel='Epochs', ylabel='Accuracy', title='Accuracy plot')
```

[39]:
```
datagen = ImageDataGenerator(featurewise_center=True,␣
 ↪featurewise_std_normalization=True, rotation_range=20,
                             fill_mode='nearest', width_shift_range=0.
 ↪2,height_shift_range=0.2,
                             vertical_flip=True, horizontal_flip=True,␣
 ↪brightness_range=[0.4,1.5], zoom_range=0.3)

datagen.fit(x_train)
```

[40]:
```python
Cnv_model_aug_layers = [

    Conv2D(32, (4, 4), strides = (2,2), padding='same', input_shape=x_train.
 ↪shape[1:]),
    Activation('relu'),

    Conv2D(32, (4, 4), strides = (2,2)),
    Activation('relu'),

    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),

    Flatten(),

    Dense(512),
    Activation('relu'),
    Dropout(0.5),
    Dense(classes),
    Activation('softmax')
]
```

[41]:
```python
Cnv_model_aug = Sequential(Cnv_model_aug_layers)
Cnv_model_aug.summary()
```

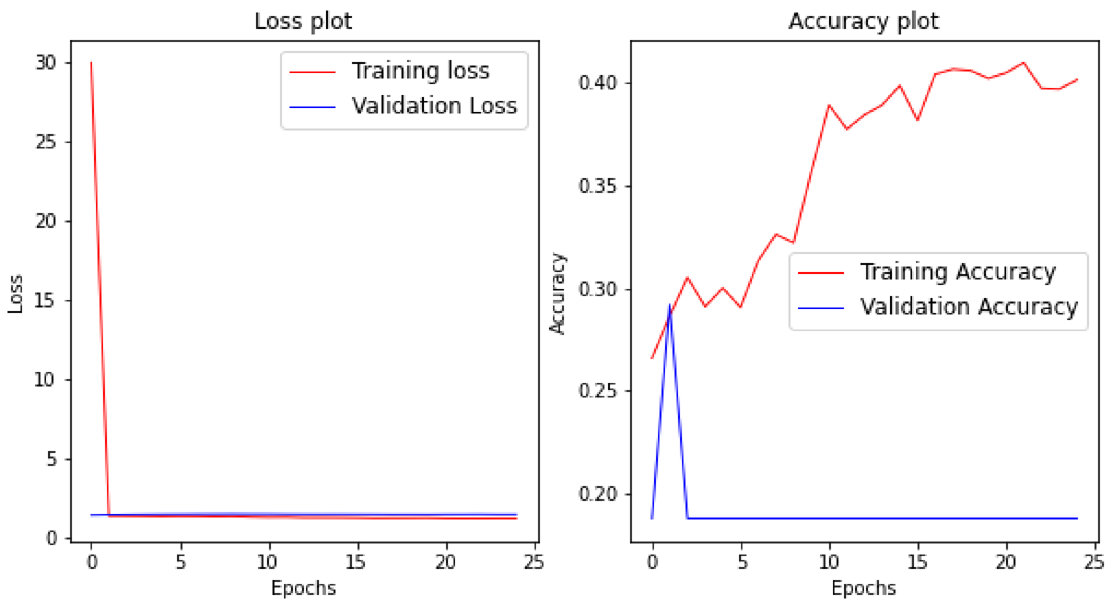
```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_4 (Conv2D)            (None, 100, 100, 32)      544
_________________________________________________________________
activation_8 (Activation)    (None, 100, 100, 32)      0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 49, 49, 32)        16416
_________________________________________________________________
activation_9 (Activation)    (None, 49, 49, 32)        0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 24, 24, 32)        0
_________________________________________________________________
dropout_4 (Dropout)          (None, 24, 24, 32)        0
_________________________________________________________________
flatten_2 (Flatten)          (None, 18432)             0
_________________________________________________________________
dense_4 (Dense)              (None, 512)               9437696
_________________________________________________________________
activation_10 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_5 (Dropout)          (None, 512)               0
```

```
_________________________________________________________________
dense_5 (Dense)              (None, 4)                 2052
_________________________________________________________________
activation_11 (Activation)   (None, 4)                 0
=================================================================
Total params: 9,456,708
Trainable params: 9,456,708
Non-trainable params: 0
_________________________________________________________________
```

[42]:
```
Cnv_model_aug.compile(loss='categorical_crossentropy',
              optimizer=opt_adam,
              metrics=['accuracy'])

m_train_aug(Cnv_model_aug, x_train, y_train, 4)
```

```
Epoch 1/25
89/89 [==============================] - 29s 323ms/step - loss: 109.4978 -
accuracy: 0.2561 - val_loss: 1.4038 - val_accuracy: 0.1878
Epoch 2/25
89/89 [==============================] - 28s 313ms/step - loss: 1.3545 -
accuracy: 0.2974 - val_loss: 1.4212 - val_accuracy: 0.2919
Epoch 3/25
89/89 [==============================] - 30s 330ms/step - loss: 1.3467 -
accuracy: 0.3071 - val_loss: 1.4353 - val_accuracy: 0.1878
Epoch 4/25
89/89 [==============================] - 28s 314ms/step - loss: 1.3415 -
accuracy: 0.3111 - val_loss: 1.4439 - val_accuracy: 0.1878
Epoch 5/25
89/89 [==============================] - 28s 317ms/step - loss: 1.3281 -
accuracy: 0.3089 - val_loss: 1.4518 - val_accuracy: 0.1878
Epoch 6/25
89/89 [==============================] - 28s 313ms/step - loss: 1.3388 -
accuracy: 0.2950 - val_loss: 1.4558 - val_accuracy: 0.1878
Epoch 7/25
89/89 [==============================] - 28s 316ms/step - loss: 1.3291 -
accuracy: 0.3128 - val_loss: 1.4629 - val_accuracy: 0.1878
Epoch 8/25
89/89 [==============================] - 28s 315ms/step - loss: 1.3185 -
accuracy: 0.3136 - val_loss: 1.4664 - val_accuracy: 0.1878
Epoch 9/25
89/89 [==============================] - 28s 316ms/step - loss: 1.3109 -
accuracy: 0.3052 - val_loss: 1.4710 - val_accuracy: 0.1878
Epoch 10/25
89/89 [==============================] - 28s 315ms/step - loss: 1.3026 -
accuracy: 0.3529 - val_loss: 1.4667 - val_accuracy: 0.1878
Epoch 11/25
89/89 [==============================] - 29s 318ms/step - loss: 1.2848 -
```

```
accuracy: 0.3883 - val_loss: 1.4642 - val_accuracy: 0.1878
Epoch 12/25
89/89 [==============================] - 28s 315ms/step - loss: 1.2798 -
accuracy: 0.3907 - val_loss: 1.4571 - val_accuracy: 0.1878
Epoch 13/25
89/89 [==============================] - 29s 321ms/step - loss: 1.2463 -
accuracy: 0.3847 - val_loss: 1.4530 - val_accuracy: 0.1878
Epoch 14/25
89/89 [==============================] - 29s 318ms/step - loss: 1.2440 -
accuracy: 0.3854 - val_loss: 1.4498 - val_accuracy: 0.1878
Epoch 15/25
89/89 [==============================] - 28s 317ms/step - loss: 1.2594 -
accuracy: 0.3867 - val_loss: 1.4514 - val_accuracy: 0.1878
Epoch 16/25
89/89 [==============================] - 29s 319ms/step - loss: 1.2592 -
accuracy: 0.3662 - val_loss: 1.4489 - val_accuracy: 0.1878
Epoch 17/25
89/89 [==============================] - 29s 325ms/step - loss: 1.2251 -
accuracy: 0.3987 - val_loss: 1.4464 - val_accuracy: 0.1878
Epoch 18/25
89/89 [==============================] - 29s 319ms/step - loss: 1.2344 -
accuracy: 0.4060 - val_loss: 1.4408 - val_accuracy: 0.1878
Epoch 19/25
89/89 [==============================] - 29s 318ms/step - loss: 1.2164 -
accuracy: 0.4074 - val_loss: 1.4427 - val_accuracy: 0.1878
Epoch 20/25
89/89 [==============================] - 29s 320ms/step - loss: 1.2324 -
accuracy: 0.4020 - val_loss: 1.4400 - val_accuracy: 0.1878
Epoch 21/25
89/89 [==============================] - 28s 316ms/step - loss: 1.2171 -
accuracy: 0.3948 - val_loss: 1.4465 - val_accuracy: 0.1878
Epoch 22/25
89/89 [==============================] - 29s 319ms/step - loss: 1.2247 -
accuracy: 0.3899 - val_loss: 1.4475 - val_accuracy: 0.1878
Epoch 23/25
89/89 [==============================] - 29s 320ms/step - loss: 1.2085 -
accuracy: 0.4014 - val_loss: 1.4514 - val_accuracy: 0.1878
Epoch 24/25
89/89 [==============================] - 29s 323ms/step - loss: 1.1976 -
accuracy: 0.3953 - val_loss: 1.4443 - val_accuracy: 0.1878
Epoch 25/25
89/89 [==============================] - 29s 326ms/step - loss: 1.1946 -
accuracy: 0.3898 - val_loss: 1.4454 - val_accuracy: 0.1878
Training time: 0:11:57.812785
Test score: 1.4453654289245605
Test accuracy: 0.1878172606229782
```



3.0.8 I THINK I NOW WHERE THE PROBLEM IS NOW !

When one uses cross-entropy loss for classification, bad predictions are penalized much more than good predictions are rewarded. Therefore, even if many images are correctly predicted, a single misclassified will have a high loss increasing significantly the mean loss. I think that when both accuracy and loss are increasing, the network is starting to overfit. The network is starting to learn patterns only relevant for the training set and not great for generalization, some images from the validation set get predicted really wrong. However, it is at the same time still learning some patterns which are useful for generalization as more and more images are being correctly classified. I think this effect can be further obscured in the case of multi-class classification, where the network at a given epoch might be severely overfit on some classes but still learning on others.

Hence I will use the MeanSquaredError loss function. I don't really know if it should or shoudln't be used in my case but it will give me an increase in validation accuracy togheter with a decrease in validation loss.

I think I can use the model used with the frist vanilla model, Cnv_model_v1, Cnv_model_jolly (a new model). However I will run it with another train function without the datagen and with more epochs

[92]:
```
def m_train32_50(model, train, test, number_classes):
    t = time()
    history = model.fit(x_train, y_train,
                        batch_size=32,
                        epochs=50, #50 eprochs
                        validation_data=(x_test, y_test),
                        shuffle=True)
    print('Training time: %s' % (time() - t))
    score = model.evaluate(x_test, y_test, verbose=0)
    print('Test score:', score[0])
    print('Test accuracy:', score[1])

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,5))

    ax1.plot(history.history['loss'],'r',linewidth=1.0)
    ax1.plot(history.history['val_loss'],'b',linewidth=1.0)
    ax1.legend(['Training loss', 'Validation Loss'],fontsize=12)
    ax1.set(xlabel='Epochs', ylabel='Loss', title='Loss plot')

    ax2.plot(history.history['accuracy'],'r',linewidth=1.0)
    ax2.plot(history.history['val_accuracy'],'b',linewidth=1.0)
    ax2.legend(['Training Accuracy', 'Validation Accuracy'],fontsize=12)
    ax2.set(xlabel='Epochs', ylabel='Accuracy', title='Accuracy plot')
```

FIRST - VANILLA MODEL

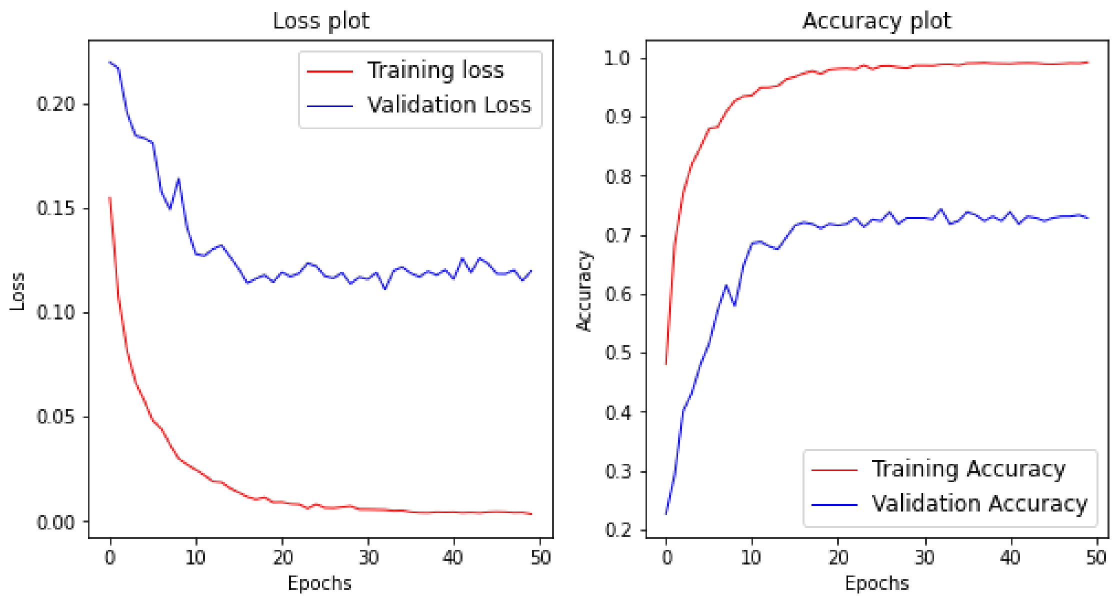
[93]:
```
Cnv_model.compile(loss='MeanSquaredError',
              optimizer=opt_adam,
              metrics=['accuracy'])

m_train32_50(Cnv_model, x_train, y_train, 4)
```

```
Epoch 1/50
90/90 [==============================] - 4s 46ms/step - loss: 0.1719 - accuracy:
0.3800 - val_loss: 0.2197 - val_accuracy: 0.2259
Epoch 2/50
90/90 [==============================] - 4s 44ms/step - loss: 0.1142 - accuracy:
0.6647 - val_loss: 0.2168 - val_accuracy: 0.2919
Epoch 3/50
90/90 [==============================] - 4s 46ms/step - loss: 0.0866 - accuracy:
0.7545 - val_loss: 0.1956 - val_accuracy: 0.4010
Epoch 4/50
90/90 [==============================] - 4s 44ms/step - loss: 0.0628 - accuracy:
0.8311 - val_loss: 0.1846 - val_accuracy: 0.4315
Epoch 5/50
90/90 [==============================] - 4s 45ms/step - loss: 0.0587 - accuracy:
0.8506 - val_loss: 0.1834 - val_accuracy: 0.4797
Epoch 6/50
90/90 [==============================] - 4s 46ms/step - loss: 0.0505 - accuracy:
0.8719 - val_loss: 0.1811 - val_accuracy: 0.5152
Epoch 7/50
90/90 [==============================] - 4s 46ms/step - loss: 0.0436 - accuracy:
0.8816 - val_loss: 0.1576 - val_accuracy: 0.5711
Epoch 8/50
90/90 [==============================] - 5s 51ms/step - loss: 0.0347 - accuracy:
```

```
0.9153 - val_loss: 0.1491 - val_accuracy: 0.6142
Epoch 9/50
90/90 [==============================] - 4s 45ms/step - loss: 0.0291 - accuracy:
0.9322 - val_loss: 0.1639 - val_accuracy: 0.5787
Epoch 10/50
90/90 [==============================] - 4s 45ms/step - loss: 0.0280 - accuracy:
0.9305 - val_loss: 0.1404 - val_accuracy: 0.6472
Epoch 11/50
90/90 [==============================] - 4s 46ms/step - loss: 0.0245 - accuracy:
0.9387 - val_loss: 0.1279 - val_accuracy: 0.6853
Epoch 12/50
90/90 [==============================] - 4s 45ms/step - loss: 0.0202 - accuracy:
0.9547 - val_loss: 0.1270 - val_accuracy: 0.6878
Epoch 13/50
90/90 [==============================] - 4s 46ms/step - loss: 0.0174 - accuracy:
0.9565 - val_loss: 0.1302 - val_accuracy: 0.6802
Epoch 14/50
90/90 [==============================] - 4s 45ms/step - loss: 0.0188 - accuracy:
0.9501 - val_loss: 0.1321 - val_accuracy: 0.6751
Epoch 15/50
90/90 [==============================] - 4s 45ms/step - loss: 0.0171 - accuracy:
0.9593 - val_loss: 0.1264 - val_accuracy: 0.6954
Epoch 16/50
90/90 [==============================] - 4s 45ms/step - loss: 0.0123 - accuracy:
0.9740 - val_loss: 0.1208 - val_accuracy: 0.7157
Epoch 17/50
90/90 [==============================] - 4s 45ms/step - loss: 0.0118 - accuracy:
0.9718 - val_loss: 0.1139 - val_accuracy: 0.7208
Epoch 18/50
90/90 [==============================] - 4s 46ms/step - loss: 0.0093 - accuracy:
0.9810 - val_loss: 0.1161 - val_accuracy: 0.7183
Epoch 19/50
90/90 [==============================] - 4s 47ms/step - loss: 0.0128 - accuracy:
0.9709 - val_loss: 0.1179 - val_accuracy: 0.7107
Epoch 20/50
90/90 [==============================] - 4s 46ms/step - loss: 0.0070 - accuracy:
0.9857 - val_loss: 0.1145 - val_accuracy: 0.7183
Epoch 21/50
90/90 [==============================] - 4s 47ms/step - loss: 0.0090 - accuracy:
0.9844 - val_loss: 0.1192 - val_accuracy: 0.7157
Epoch 22/50
90/90 [==============================] - 4s 46ms/step - loss: 0.0071 - accuracy:
0.9863 - val_loss: 0.1171 - val_accuracy: 0.7183
Epoch 23/50
90/90 [==============================] - 4s 47ms/step - loss: 0.0077 - accuracy:
0.9815 - val_loss: 0.1186 - val_accuracy: 0.7284
Epoch 24/50
90/90 [==============================] - 4s 45ms/step - loss: 0.0050 - accuracy:
```

```
0.9904 - val_loss: 0.1235 - val_accuracy: 0.7132
Epoch 25/50
90/90 [==============================] - 4s 46ms/step - loss: 0.0090 - accuracy:
0.9788 - val_loss: 0.1221 - val_accuracy: 0.7259
Epoch 26/50
90/90 [==============================] - 4s 46ms/step - loss: 0.0068 - accuracy:
0.9853 - val_loss: 0.1174 - val_accuracy: 0.7234
Epoch 27/50
90/90 [==============================] - 4s 46ms/step - loss: 0.0064 - accuracy:
0.9861 - val_loss: 0.1163 - val_accuracy: 0.7386
Epoch 28/50
90/90 [==============================] - 4s 45ms/step - loss: 0.0066 - accuracy:
0.9850 - val_loss: 0.1190 - val_accuracy: 0.7183
Epoch 29/50
90/90 [==============================] - 4s 46ms/step - loss: 0.0074 - accuracy:
0.9829 - val_loss: 0.1136 - val_accuracy: 0.7284
Epoch 30/50
90/90 [==============================] - 4s 46ms/step - loss: 0.0061 - accuracy:
0.9827 - val_loss: 0.1169 - val_accuracy: 0.7284
Epoch 31/50
90/90 [==============================] - 4s 45ms/step - loss: 0.0056 - accuracy:
0.9878 - val_loss: 0.1159 - val_accuracy: 0.7284
Epoch 32/50
90/90 [==============================] - 4s 45ms/step - loss: 0.0058 - accuracy:
0.9850 - val_loss: 0.1191 - val_accuracy: 0.7259
Epoch 33/50
90/90 [==============================] - 4s 45ms/step - loss: 0.0051 - accuracy:
0.9897 - val_loss: 0.1109 - val_accuracy: 0.7437
Epoch 34/50
90/90 [==============================] - 4s 49ms/step - loss: 0.0046 - accuracy:
0.9905 - val_loss: 0.1199 - val_accuracy: 0.7183
Epoch 35/50
90/90 [==============================] - 5s 50ms/step - loss: 0.0059 - accuracy:
0.9848 - val_loss: 0.1217 - val_accuracy: 0.7234
Epoch 36/50
90/90 [==============================] - 4s 49ms/step - loss: 0.0034 - accuracy:
0.9931 - val_loss: 0.1187 - val_accuracy: 0.7386
Epoch 37/50
90/90 [==============================] - 4s 47ms/step - loss: 0.0037 - accuracy:
0.9932 - val_loss: 0.1170 - val_accuracy: 0.7335
Epoch 38/50
90/90 [==============================] - 4s 47ms/step - loss: 0.0039 - accuracy:
0.9929 - val_loss: 0.1196 - val_accuracy: 0.7234
Epoch 39/50
90/90 [==============================] - 4s 46ms/step - loss: 0.0039 - accuracy:
0.9903 - val_loss: 0.1178 - val_accuracy: 0.7310
Epoch 40/50
90/90 [==============================] - 4s 46ms/step - loss: 0.0047 - accuracy:
```

```
0.9885 - val_loss: 0.1204 - val_accuracy: 0.7234
Epoch 41/50
90/90 [==============================] - 4s 47ms/step - loss: 0.0044 - accuracy:
0.9876 - val_loss: 0.1159 - val_accuracy: 0.7386
Epoch 42/50
90/90 [==============================] - 4s 46ms/step - loss: 0.0034 - accuracy:
0.9930 - val_loss: 0.1258 - val_accuracy: 0.7183
Epoch 43/50
90/90 [==============================] - 4s 47ms/step - loss: 0.0045 - accuracy:
0.9892 - val_loss: 0.1190 - val_accuracy: 0.7310
Epoch 44/50
90/90 [==============================] - 4s 46ms/step - loss: 0.0034 - accuracy:
0.9922 - val_loss: 0.1258 - val_accuracy: 0.7284
Epoch 45/50
90/90 [==============================] - 4s 50ms/step - loss: 0.0036 - accuracy:
0.9907 - val_loss: 0.1232 - val_accuracy: 0.7234
Epoch 46/50
90/90 [==============================] - 4s 46ms/step - loss: 0.0055 - accuracy:
0.9855 - val_loss: 0.1185 - val_accuracy: 0.7284
Epoch 47/50
90/90 [==============================] - 4s 47ms/step - loss: 0.0038 - accuracy:
0.9906 - val_loss: 0.1184 - val_accuracy: 0.7310
Epoch 48/50
90/90 [==============================] - 4s 45ms/step - loss: 0.0032 - accuracy:
0.9935 - val_loss: 0.1202 - val_accuracy: 0.7310
Epoch 49/50
90/90 [==============================] - 4s 46ms/step - loss: 0.0040 - accuracy:
0.9917 - val_loss: 0.1151 - val_accuracy: 0.7335
Epoch 50/50
90/90 [==============================] - 4s 46ms/step - loss: 0.0044 - accuracy:
0.9902 - val_loss: 0.1198 - val_accuracy: 0.7284
Training time: 0:03:28.384329
Test score: 0.11975941807031631
Test accuracy: 0.7284263968467712
```



CNV_MODEL_V1

```
[94]: Cnv_model_v1.compile(loss='MeanSquaredError',
                     optimizer=opt_adam,
                     metrics=['accuracy'])

      m_train32_50(Cnv_model_v1, x_train, y_train, 4)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-94-bafee9acd9b4> in <module>
----> 1 Cnv_model_v1.compile(loss='MeanSquaredError',

      2                  optimizer=opt_adam,
      3                  metrics=['accuracy'])
      4
      5 m_train32_50(Cnv_model_v1, x_train, y_train, 4)

NameError: name 'Cnv_model_v1' is not defined
```

CONV_VERSION JOLLY

```
[95]: Cnv_model_jolly_layers = [
          Conv2D(32, (5, 5), strides = (2,2), padding='same', input_shape=x_train.
      ↪shape[1:]),
          Activation('relu'),
          Conv2D(32, (5, 5), strides = (2,2)),
          Activation('relu'),

          MaxPooling2D(pool_size=(2, 2)),
          Dropout(0.25),

          Conv2D(32, (5, 5), strides = (2,2), padding='same', input_shape=x_train.
       ↪shape[1:]),
          Activation('relu'),
          Conv2D(32, (5, 5), strides = (2,2)),
          Activation('relu'),

          MaxPooling2D(pool_size=(2, 2)),
          Dropout(0.25),

          Flatten(),

          Dense(512),
          Activation('relu'),
          Dropout(0.5),
          Dense(classes),
          Activation('softmax')
      ]
```

[96]:
```
Cnv_model_jolly = Sequential(Cnv_model_jolly_layers)
```

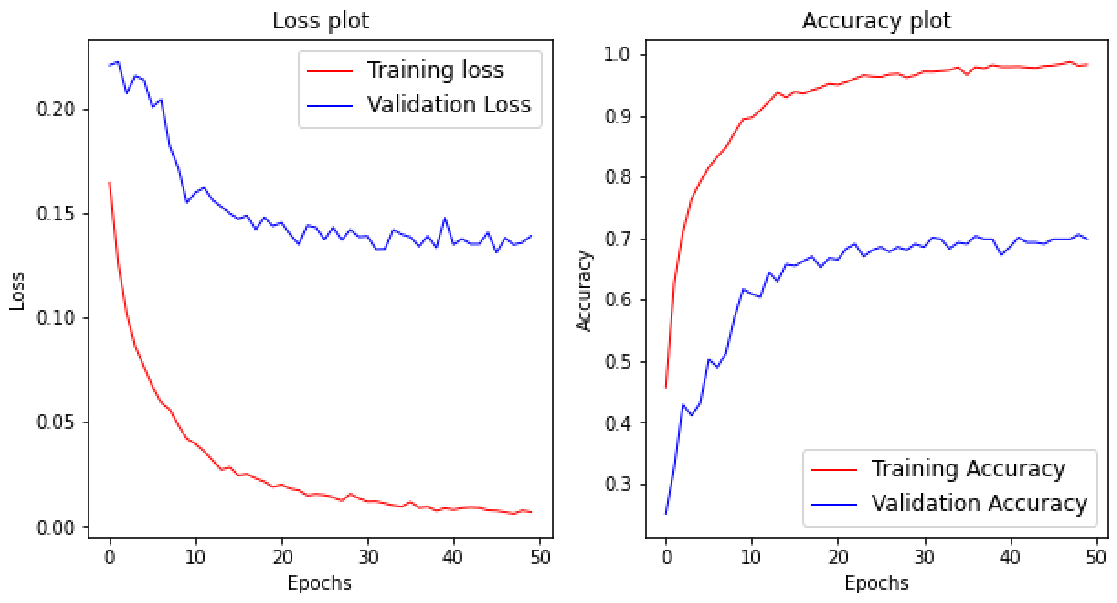
[97]:
```
Cnv_model_jolly.compile(loss='MeanSquaredError',
              optimizer=opt_adam,
              metrics=['accuracy'])

m_train32_50(Cnv_model_jolly, x_train, y_train, 4)
```

```
Epoch 1/50
90/90 [==============================] - 30s 323ms/step - loss: 0.1756 -
accuracy: 0.3799 - val_loss: 0.2206 - val_accuracy: 0.2513
Epoch 2/50
90/90 [==============================] - 29s 320ms/step - loss: 0.1349 -
accuracy: 0.5892 - val_loss: 0.2222 - val_accuracy: 0.3274
Epoch 3/50
90/90 [==============================] - 29s 325ms/step - loss: 0.0989 -
accuracy: 0.7237 - val_loss: 0.2072 - val_accuracy: 0.4289
Epoch 4/50
90/90 [==============================] - 30s 332ms/step - loss: 0.0851 -
accuracy: 0.7657 - val_loss: 0.2156 - val_accuracy: 0.4112
Epoch 5/50
90/90 [==============================] - 29s 327ms/step - loss: 0.0766 -
accuracy: 0.7910 - val_loss: 0.2137 - val_accuracy: 0.4315
Epoch 6/50
```

```
90/90 [==============================] - 29s 321ms/step - loss: 0.0689 -
accuracy: 0.8045 - val_loss: 0.2006 - val_accuracy: 0.5025
Epoch 7/50
90/90 [==============================] - 29s 324ms/step - loss: 0.0579 -
accuracy: 0.8335 - val_loss: 0.2042 - val_accuracy: 0.4898
Epoch 8/50
90/90 [==============================] - 29s 323ms/step - loss: 0.0537 -
accuracy: 0.8515 - val_loss: 0.1819 - val_accuracy: 0.5127
Epoch 9/50
90/90 [==============================] - 30s 331ms/step - loss: 0.0473 -
accuracy: 0.8766 - val_loss: 0.1716 - val_accuracy: 0.5711
Epoch 10/50
90/90 [==============================] - 30s 334ms/step - loss: 0.0419 -
accuracy: 0.8917 - val_loss: 0.1549 - val_accuracy: 0.6168
Epoch 11/50
90/90 [==============================] - 30s 333ms/step - loss: 0.0376 -
accuracy: 0.9035 - val_loss: 0.1597 - val_accuracy: 0.6091
Epoch 12/50
90/90 [==============================] - 29s 322ms/step - loss: 0.0336 -
accuracy: 0.9175 - val_loss: 0.1621 - val_accuracy: 0.6041
Epoch 13/50
90/90 [==============================] - 29s 320ms/step - loss: 0.0336 -
accuracy: 0.9184 - val_loss: 0.1561 - val_accuracy: 0.6447
Epoch 14/50
90/90 [==============================] - 29s 321ms/step - loss: 0.0286 -
accuracy: 0.9347 - val_loss: 0.1530 - val_accuracy: 0.6294
Epoch 15/50
90/90 [==============================] - 29s 324ms/step - loss: 0.0271 -
accuracy: 0.9342 - val_loss: 0.1497 - val_accuracy: 0.6574
Epoch 16/50
90/90 [==============================] - 29s 324ms/step - loss: 0.0225 -
accuracy: 0.9417 - val_loss: 0.1471 - val_accuracy: 0.6548
Epoch 17/50
90/90 [==============================] - 29s 323ms/step - loss: 0.0225 -
accuracy: 0.9417 - val_loss: 0.1488 - val_accuracy: 0.6624
Epoch 18/50
90/90 [==============================] - 29s 323ms/step - loss: 0.0212 -
accuracy: 0.9465 - val_loss: 0.1422 - val_accuracy: 0.6701
Epoch 19/50
90/90 [==============================] - 29s 326ms/step - loss: 0.0229 -
accuracy: 0.9448 - val_loss: 0.1479 - val_accuracy: 0.6523
Epoch 20/50
90/90 [==============================] - 29s 323ms/step - loss: 0.0179 -
accuracy: 0.9549 - val_loss: 0.1438 - val_accuracy: 0.6675
Epoch 21/50
90/90 [==============================] - 29s 318ms/step - loss: 0.0193 -
accuracy: 0.9508 - val_loss: 0.1454 - val_accuracy: 0.6650
Epoch 22/50
```

```
90/90 [==============================] - 29s 321ms/step - loss: 0.0186 -
accuracy: 0.9524 - val_loss: 0.1396 - val_accuracy: 0.6827
Epoch 23/50
90/90 [==============================] - 29s 318ms/step - loss: 0.0185 -
accuracy: 0.9539 - val_loss: 0.1350 - val_accuracy: 0.6904
Epoch 24/50
90/90 [==============================] - 29s 322ms/step - loss: 0.0156 -
accuracy: 0.9662 - val_loss: 0.1440 - val_accuracy: 0.6701
Epoch 25/50
90/90 [==============================] - 29s 318ms/step - loss: 0.0158 -
accuracy: 0.9625 - val_loss: 0.1430 - val_accuracy: 0.6802
Epoch 26/50
90/90 [==============================] - 29s 320ms/step - loss: 0.0162 -
accuracy: 0.9603 - val_loss: 0.1372 - val_accuracy: 0.6853
Epoch 27/50
90/90 [==============================] - 29s 319ms/step - loss: 0.0121 -
accuracy: 0.9736 - val_loss: 0.1432 - val_accuracy: 0.6777
Epoch 28/50
90/90 [==============================] - 29s 319ms/step - loss: 0.0139 -
accuracy: 0.9651 - val_loss: 0.1370 - val_accuracy: 0.6853
Epoch 29/50
90/90 [==============================] - 29s 320ms/step - loss: 0.0134 -
accuracy: 0.9653 - val_loss: 0.1420 - val_accuracy: 0.6802
Epoch 30/50
90/90 [==============================] - 29s 325ms/step - loss: 0.0140 -
accuracy: 0.9598 - val_loss: 0.1385 - val_accuracy: 0.6904
Epoch 31/50
90/90 [==============================] - 29s 318ms/step - loss: 0.0110 -
accuracy: 0.9738 - val_loss: 0.1388 - val_accuracy: 0.6853
Epoch 32/50
90/90 [==============================] - 29s 321ms/step - loss: 0.0115 -
accuracy: 0.9719 - val_loss: 0.1325 - val_accuracy: 0.7005
Epoch 33/50
90/90 [==============================] - 29s 319ms/step - loss: 0.0103 -
accuracy: 0.9741 - val_loss: 0.1326 - val_accuracy: 0.6980
Epoch 34/50
90/90 [==============================] - 29s 319ms/step - loss: 0.0092 -
accuracy: 0.9759 - val_loss: 0.1420 - val_accuracy: 0.6827
Epoch 35/50
90/90 [==============================] - 29s 318ms/step - loss: 0.0100 -
accuracy: 0.9778 - val_loss: 0.1397 - val_accuracy: 0.6929
Epoch 36/50
90/90 [==============================] - 29s 317ms/step - loss: 0.0118 -
accuracy: 0.9658 - val_loss: 0.1384 - val_accuracy: 0.6904
Epoch 37/50
90/90 [==============================] - 29s 320ms/step - loss: 0.0094 -
accuracy: 0.9787 - val_loss: 0.1340 - val_accuracy: 0.7030
Epoch 38/50
```

```
90/90 [==============================] - 29s 320ms/step - loss: 0.0094 -
accuracy: 0.9763 - val_loss: 0.1389 - val_accuracy: 0.6980
Epoch 39/50
90/90 [==============================] - 29s 320ms/step - loss: 0.0084 -
accuracy: 0.9770 - val_loss: 0.1334 - val_accuracy: 0.6980
Epoch 40/50
90/90 [==============================] - 29s 319ms/step - loss: 0.0101 -
accuracy: 0.9763 - val_loss: 0.1476 - val_accuracy: 0.6726
Epoch 41/50
90/90 [==============================] - 29s 318ms/step - loss: 0.0065 -
accuracy: 0.9835 - val_loss: 0.1349 - val_accuracy: 0.6853
Epoch 42/50
90/90 [==============================] - 29s 326ms/step - loss: 0.0097 -
accuracy: 0.9760 - val_loss: 0.1376 - val_accuracy: 0.7005
Epoch 43/50
90/90 [==============================] - 29s 321ms/step - loss: 0.0094 -
accuracy: 0.9775 - val_loss: 0.1351 - val_accuracy: 0.6929
Epoch 44/50
90/90 [==============================] - 29s 321ms/step - loss: 0.0089 -
accuracy: 0.9768 - val_loss: 0.1351 - val_accuracy: 0.6929
Epoch 45/50
90/90 [==============================] - 29s 322ms/step - loss: 0.0065 -
accuracy: 0.9824 - val_loss: 0.1407 - val_accuracy: 0.6904
Epoch 46/50
90/90 [==============================] - 29s 323ms/step - loss: 0.0078 -
accuracy: 0.9793 - val_loss: 0.1310 - val_accuracy: 0.6980
Epoch 47/50
90/90 [==============================] - 29s 322ms/step - loss: 0.0072 -
accuracy: 0.9828 - val_loss: 0.1380 - val_accuracy: 0.6980
Epoch 48/50
90/90 [==============================] - 29s 322ms/step - loss: 0.0056 -
accuracy: 0.9879 - val_loss: 0.1348 - val_accuracy: 0.6980
Epoch 49/50
90/90 [==============================] - 29s 323ms/step - loss: 0.0064 -
accuracy: 0.9838 - val_loss: 0.1359 - val_accuracy: 0.7056
Epoch 50/50
90/90 [==============================] - 29s 320ms/step - loss: 0.0080 -
accuracy: 0.9799 - val_loss: 0.1389 - val_accuracy: 0.6980
Training time: 0:24:10.233510
Test score: 0.1388760507106781
Test accuracy: 0.6979695558547974
```



AUGMENTED MODEL

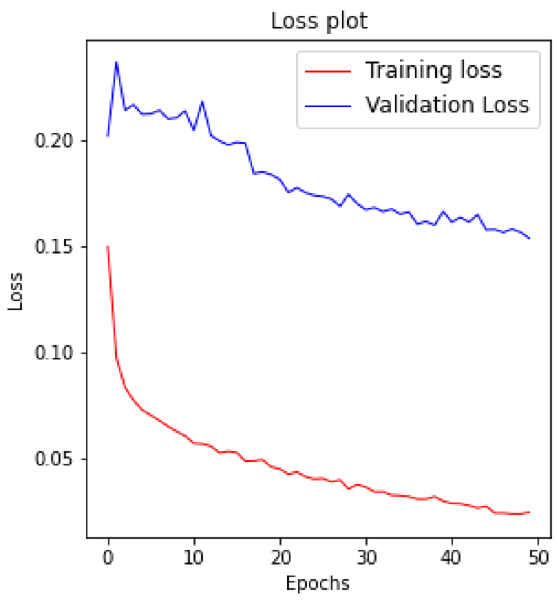
[98]:
```
Cnv_model_aug.compile(loss='MeanSquaredError',
                      optimizer=opt_adam,
                      metrics=['accuracy'])

m_train32_50(Cnv_model_aug, x_train, y_train, 4)
```

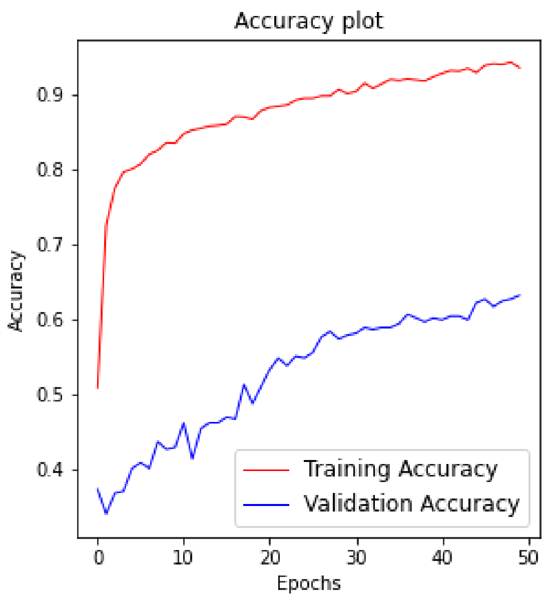
```
Epoch 1/50
90/90 [==============================] - 27s 291ms/step - loss: 0.1634 -
accuracy: 0.4499 - val_loss: 0.2020 - val_accuracy: 0.3731
Epoch 2/50
90/90 [==============================] - 26s 293ms/step - loss: 0.1056 -
accuracy: 0.6998 - val_loss: 0.2369 - val_accuracy: 0.3401
Epoch 3/50
90/90 [==============================] - 26s 293ms/step - loss: 0.0875 -
accuracy: 0.7581 - val_loss: 0.2142 - val_accuracy: 0.3680
Epoch 4/50
90/90 [==============================] - 27s 295ms/step - loss: 0.0761 -
accuracy: 0.7982 - val_loss: 0.2168 - val_accuracy: 0.3706
Epoch 5/50
90/90 [==============================] - 27s 295ms/step - loss: 0.0725 -
accuracy: 0.8022 - val_loss: 0.2122 - val_accuracy: 0.4010
Epoch 6/50
90/90 [==============================] - 27s 300ms/step - loss: 0.0683 -
accuracy: 0.8171 - val_loss: 0.2125 - val_accuracy: 0.4086
Epoch 7/50
90/90 [==============================] - 26s 294ms/step - loss: 0.0673 -
```

```
accuracy: 0.8188 - val_loss: 0.2141 - val_accuracy: 0.4010
Epoch 8/50
90/90 [==============================] - 27s 296ms/step - loss: 0.0628 -
accuracy: 0.8344 - val_loss: 0.2101 - val_accuracy: 0.4365
Epoch 9/50
90/90 [==============================] - 27s 296ms/step - loss: 0.0617 -
accuracy: 0.8379 - val_loss: 0.2106 - val_accuracy: 0.4264
Epoch 10/50
90/90 [==============================] - 27s 297ms/step - loss: 0.0607 -
accuracy: 0.8320 - val_loss: 0.2138 - val_accuracy: 0.4289
Epoch 11/50
90/90 [==============================] - 27s 295ms/step - loss: 0.0561 -
accuracy: 0.8539 - val_loss: 0.2046 - val_accuracy: 0.4619
Epoch 12/50
90/90 [==============================] - 26s 293ms/step - loss: 0.0550 -
accuracy: 0.8596 - val_loss: 0.2182 - val_accuracy: 0.4137
Epoch 13/50
90/90 [==============================] - 27s 297ms/step - loss: 0.0552 -
accuracy: 0.8553 - val_loss: 0.2022 - val_accuracy: 0.4543
Epoch 14/50
90/90 [==============================] - 27s 297ms/step - loss: 0.0506 -
accuracy: 0.8674 - val_loss: 0.1997 - val_accuracy: 0.4619
Epoch 15/50
90/90 [==============================] - 27s 298ms/step - loss: 0.0530 -
accuracy: 0.8613 - val_loss: 0.1978 - val_accuracy: 0.4619
Epoch 16/50
90/90 [==============================] - 27s 297ms/step - loss: 0.0514 -
accuracy: 0.8618 - val_loss: 0.1990 - val_accuracy: 0.4695
Epoch 17/50
90/90 [==============================] - 27s 300ms/step - loss: 0.0499 -
accuracy: 0.8664 - val_loss: 0.1986 - val_accuracy: 0.4670
Epoch 18/50
90/90 [==============================] - 27s 297ms/step - loss: 0.0496 -
accuracy: 0.8639 - val_loss: 0.1842 - val_accuracy: 0.5127
Epoch 19/50
90/90 [==============================] - 27s 297ms/step - loss: 0.0488 -
accuracy: 0.8670 - val_loss: 0.1850 - val_accuracy: 0.4873
Epoch 20/50
90/90 [==============================] - 27s 297ms/step - loss: 0.0453 -
accuracy: 0.8779 - val_loss: 0.1837 - val_accuracy: 0.5102
Epoch 21/50
90/90 [==============================] - 27s 295ms/step - loss: 0.0489 -
accuracy: 0.8693 - val_loss: 0.1815 - val_accuracy: 0.5330
Epoch 22/50
90/90 [==============================] - 27s 297ms/step - loss: 0.0444 -
accuracy: 0.8778 - val_loss: 0.1753 - val_accuracy: 0.5482
Epoch 23/50
90/90 [==============================] - 27s 296ms/step - loss: 0.0412 -
```

```
accuracy: 0.8953 - val_loss: 0.1776 - val_accuracy: 0.5381
Epoch 24/50
90/90 [==============================] - 27s 296ms/step - loss: 0.0414 -
accuracy: 0.8960 - val_loss: 0.1753 - val_accuracy: 0.5508
Epoch 25/50
90/90 [==============================] - 27s 295ms/step - loss: 0.0376 -
accuracy: 0.9020 - val_loss: 0.1740 - val_accuracy: 0.5482
Epoch 26/50
90/90 [==============================] - 27s 296ms/step - loss: 0.0415 -
accuracy: 0.8884 - val_loss: 0.1734 - val_accuracy: 0.5558
Epoch 27/50
90/90 [==============================] - 27s 296ms/step - loss: 0.0379 -
accuracy: 0.9027 - val_loss: 0.1723 - val_accuracy: 0.5761
Epoch 28/50
90/90 [==============================] - 27s 300ms/step - loss: 0.0402 -
accuracy: 0.8967 - val_loss: 0.1688 - val_accuracy: 0.5838
Epoch 29/50
90/90 [==============================] - 27s 296ms/step - loss: 0.0340 -
accuracy: 0.9100 - val_loss: 0.1744 - val_accuracy: 0.5736
Epoch 30/50
90/90 [==============================] - 27s 295ms/step - loss: 0.0398 -
accuracy: 0.8916 - val_loss: 0.1701 - val_accuracy: 0.5787
Epoch 31/50
90/90 [==============================] - 27s 296ms/step - loss: 0.0357 -
accuracy: 0.9050 - val_loss: 0.1672 - val_accuracy: 0.5812
Epoch 32/50
90/90 [==============================] - 27s 299ms/step - loss: 0.0348 -
accuracy: 0.9123 - val_loss: 0.1682 - val_accuracy: 0.5888
Epoch 33/50
90/90 [==============================] - 27s 300ms/step - loss: 0.0376 -
accuracy: 0.8982 - val_loss: 0.1663 - val_accuracy: 0.5863
Epoch 34/50
90/90 [==============================] - 29s 325ms/step - loss: 0.0346 -
accuracy: 0.9079 - val_loss: 0.1675 - val_accuracy: 0.5888
Epoch 35/50
90/90 [==============================] - 27s 305ms/step - loss: 0.0325 -
accuracy: 0.9167 - val_loss: 0.1652 - val_accuracy: 0.5888
Epoch 36/50
90/90 [==============================] - 27s 298ms/step - loss: 0.0320 -
accuracy: 0.9185 - val_loss: 0.1662 - val_accuracy: 0.5939
Epoch 37/50
90/90 [==============================] - 27s 299ms/step - loss: 0.0304 -
accuracy: 0.9231 - val_loss: 0.1604 - val_accuracy: 0.6066
Epoch 38/50
90/90 [==============================] - 27s 301ms/step - loss: 0.0297 -
accuracy: 0.9225 - val_loss: 0.1617 - val_accuracy: 0.6015
Epoch 39/50
90/90 [==============================] - 27s 301ms/step - loss: 0.0318 -
```

```
accuracy: 0.9184 - val_loss: 0.1599 - val_accuracy: 0.5964
Epoch 40/50
90/90 [==============================] - 27s 300ms/step - loss: 0.0295 -
accuracy: 0.9246 - val_loss: 0.1664 - val_accuracy: 0.6015
Epoch 41/50
90/90 [==============================] - 27s 297ms/step - loss: 0.0281 -
accuracy: 0.9288 - val_loss: 0.1613 - val_accuracy: 0.5990
Epoch 42/50
90/90 [==============================] - 27s 300ms/step - loss: 0.0276 -
accuracy: 0.9345 - val_loss: 0.1637 - val_accuracy: 0.6041
Epoch 43/50
90/90 [==============================] - 27s 299ms/step - loss: 0.0253 -
accuracy: 0.9345 - val_loss: 0.1614 - val_accuracy: 0.6041
Epoch 44/50
90/90 [==============================] - 27s 297ms/step - loss: 0.0270 -
accuracy: 0.9319 - val_loss: 0.1649 - val_accuracy: 0.5990
Epoch 45/50
90/90 [==============================] - 27s 297ms/step - loss: 0.0272 -
accuracy: 0.9304 - val_loss: 0.1578 - val_accuracy: 0.6218
Epoch 46/50
90/90 [==============================] - 27s 297ms/step - loss: 0.0212 -
accuracy: 0.9472 - val_loss: 0.1579 - val_accuracy: 0.6269
Epoch 47/50
90/90 [==============================] - 27s 303ms/step - loss: 0.0235 -
accuracy: 0.9421 - val_loss: 0.1565 - val_accuracy: 0.6168
Epoch 48/50
90/90 [==============================] - 27s 295ms/step - loss: 0.0237 -
accuracy: 0.9400 - val_loss: 0.1581 - val_accuracy: 0.6244
Epoch 49/50
90/90 [==============================] - 27s 297ms/step - loss: 0.0239 -
accuracy: 0.9440 - val_loss: 0.1566 - val_accuracy: 0.6269
Epoch 50/50
90/90 [==============================] - 27s 299ms/step - loss: 0.0250 -
accuracy: 0.9337 - val_loss: 0.1537 - val_accuracy: 0.6320
Training time: 0:22:19.607292
Test score: 0.15373027324676514
Test accuracy: 0.6319797039031982
```


Loss plot
Training loss
Validation Loss
Loss
Epochs
0.20
0.15
0.10
0.05
0
10
20
30
40
50


Accuracy plot
Training Accuracy
Validation Accuracy
Accuracy
Epochs
0.9
0.8
0.7
0.6
0.5
0.4
0
10
20
30
40
50