Athanasopoulos Nikolaos

# Assignment 2: Deep Learning with CNNs

Computer Vision

# Table of Contents

# System Hardware

The training of all models was done on a GPU:

GPU Model: Nvidia GTX 970

1664 Cuda Cores running on Cuda 11.3

4GB GDDR5 Memory

200 Watt Power Draw during training

Total training time: 15 hours

# Step 1 Framework

The framework selected for this assignment was Pytorch as it was the easiest to configure the model training on GPU.

# Step 2 Dataset

The dataset selected was FER2013 (downloaded from the assignment link) as it was easy enough to handle and on which to train multiple models without requiring too much computational power.

# Step 3 Parsing, Preprocessing and Loading

The preprocessing techniques applied aim to improve the overall performance of the model and increase its generalization ability in the final steps of the assignment. These techniques are the following:

First, we removed all the pictures for which the highest pixel value was 0, which were empty black images.

Then we split the data to training, public test and private testing according to the respective tags in the dataset.

After plotting the number of entries for each of the 7 emotions, we discovered that the dataset was imbalanced: the class “Disgust” has less than 500 entries while other emotions have over 7000 entries. This would of course create performance issues in the training phase of each model, so we decided to balance the dataset by performing oversampling, i.e. duplicating images in the classes in order for all classes to have equal amounts of entries.

Then we created the augmentation techniques and their respective classes which would be applied to the data before loading them into the model. These augmentation techniques are:

- Random rotation by at most 30 degrees
- Random warp by at most 7 pixels at each axis
- Random flip (left to right or right to left)
- Random Gaussian blur (with sigma from 0 to 0.8)
- Random noise insertion after normalizing the image

The final preprocessing technique is reshaping each image to (1, 48, 48) for transforming it to a Pytorch Tensor, and then normalizing each image to the range [0,1] for more robust results (taking pixel intensity away from the parameters).

Now our dataset is complete and we are ready to load images to the model, after creating it and specifying its parameters.

The batch size was set to 64, because any higher number resulted in memory errors for our system.

# Steps 4 and 5 Building and Testing different CNN Models

The most important part of this assignment was to test many different parameters and architectures and see which ones could lead to the best result, and so this is exactly what was performed:

9 different architectures were initially considered, all of which contained different number of layers, convolution kernel sizes, batch normalizations, fully connected layers, max pooling and dropouts. We made sure to test a lot of different architectures and changed a lot of layers to make sure that we could find the best possible architecture for our final model.

For these 9 architectures, we ran the training on RMSProp optimizer. Then, for the 9 same models, we ran the training again but with Adam optimizer, in order to test which optimizer leads to a better model. These 9 Adam optimized models are annotated with V2 on their name.

Then, for the 3 best performing models of the previous 18 (9 RMSProp + 9 Adam) we changed the initial learning rate from 1e-3 to 1e-4 and 1e-2. These 3 + 3 models are annotated with V3 and V4 respectively.

So all in all, we have 24 trained models, all of which give valuable information as to which set of parameters and architecture leads to the best results for out multi class classification problem.

It should be noted that for all models (except model v3_mml where Multi Margin Loss was used just for testing) the loss function “cross entropy” was used as it is the best one for multi class classification problems such as ours.

A table showing the models’ metrics and various parameters can be seen here:

| Model | Approximate Average time/epoch (seconds) | Epochs Ran | Number of Trainable Parameters | Training Accuracy After All epochs (%) | Validation Accuracy After All epochs (%) | Test Accuracy (%) | Test Precision (%) | Test Recall (%) | Test f1 (%) |
|---|---|---|---|---|---|---|---|---|---|
| MODEL1 | 120 | 20 | 6553991 | 61,61 | 55,38 | 56,56 | 51,87 | 59,09 | 51,45 |
| MODEL1V2 | 120 | 20 | 6553991 | 60,08 | 54,35 | 54,72 | 50,35 | 57,65 | 49,81 |
| MODEL1V3 | 120 | 20 | 6553991 | 62,19 | 54,35 | 56,90 | 52,13 | 60,62 | 51,73 |
| MODEL2 | 120 | 20 | 6553991 | 57,40 | 52,29 | 54,00 | 49,96 | 56,83 | 48,65 |
| MODEL2V2 | 120 | 20 | 6553991 | 58,72 | 53,32 | 54,17 | 49,77 | 57,18 | 48,99 |
| MODEL3 | 80 | 20 | 2130311 | 62,94 | 54,54 | 55,36 | 50,85 | 58,01 | 50,43 |
| MODEL3V2 | 120 | 20 | 2130311 | 62,54 | 54,12 | 56,95 | 52,62 | 59,50 | 52,31 |
| MODEL3V3 | 120 | 20 | 2130311 | 61,07 | 52,93 | 53,78 | 49,35 | 56,65 | 49,26 |
| MODEL3V3_MML | 120 | 20 | 2130311 | 57,66 | 52,70 | 53,75 | 48,54 | 55,85 | 48,30 |
| MODEL3V4 | 120 | 20 | 2130311 | 56,34 | 50,70 | 51,46 | 47,32 | 53,41 | 46,80 |
| MODEL4 | 80 | 20 | 2130311 | 57,91 | 50,67 | 52,61 | 49,37 | 56,02 | 48,06 |
| MODEL4V2 | 80 | 20 | 2130311 | 58,20 | 52,01 | 54,17 | 50,54 | 57,43 | 49,43 |
| MODEL5 | 80 | 20 | 314023 | 48,06 | 47,85 | 49,57 | 44,36 | 51,54 | 43,13 |
| MODEL5V2 | 80 | 20 | 314023 | 46,84 | 47,05 | 47,48 | 42,83 | 50,23 | 41,54 |
| MODEL6 | 120 | 20 | 2393543 | 56,10 | 52,06 | 53,44 | 48,87 | 57,00 | 48,38 |
| MODEL6V2 | 110 | 20 | 2393543 | 56,50 | 52,79 | 54,36 | 49,17 | 57,22 | 49,03 |
| MODEL7 | 100 | 20 | 6783559 | 63,63 | 56,24 | 56,87 | 52,07 | 59,50 | 52,03 |
| MODEL7V2 | 100 | 20 | 6783559 | 62,83 | 55,57 | 56,84 | 51,73 | 59,79 | 52,00 |
| MODEL7V3 | 100 | 20 | 6783559 | 64,94 | 55,18 | 57,15 | 52,55 | 60,46 | 52,43 |
| MODEL7V4 | 100 | 20 | 6783559 | 56,72 | 49,97 | 51,49 | 48,33 | 55,07 | 46,88 |
| MODEL8 | 100 | 20 | 7836039 | 59,95 | 54,99 | 55,42 | 50,92 | 58,35 | 50,68 |
| MODEL8V2 | 100 | 20 | 7836039 | 58,27 | 52,31 | 54,58 | 50,09 | 57,20 | 49,69 |
| MODEL9 | 160 | 10 | 6634343 | 52,93 | 50,33 | 50,99 | 46,16 | 53,00 | 45,25 |
| MODEL9V2 | 160 | 10 | 6634343 | 52,68 | 48,66 | 50,79 | 47,05 | 53,71 | 45,51 |

| | |
|---|---|
| **TOTAL SECONDS SPENT ON TRAINING** | 51800 |
| **IN MINUTES** | 863,33 |
| **IN HOURS** | 14,38 |

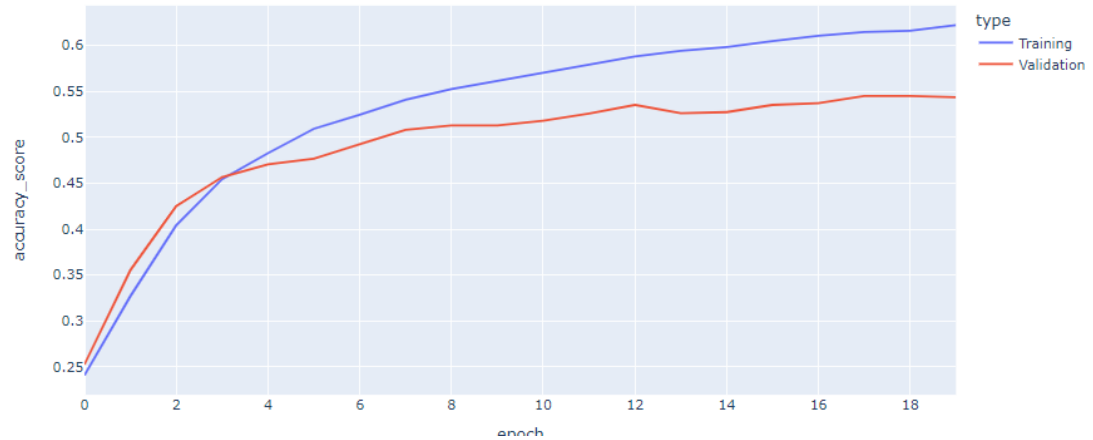
It should be noted that we chose to run most models for 20 epochs, because in our preliminary testing this seemed to be the average number of epochs after which the training and validation accuracies diverge from each other, which means that the model is over-fitting and thus the training has to be stopped. For some of our models, these accuracies do not seem to diverge and this means that we should have let these models train for some epochs more, to see if this would yield much better results (although in most of these cases, the accuracies plot seem to be very close to horizontal after all epochs which means that the modes have reached almost their maximum accuracy).

The training and accuracy plots for every model trained are presented here:

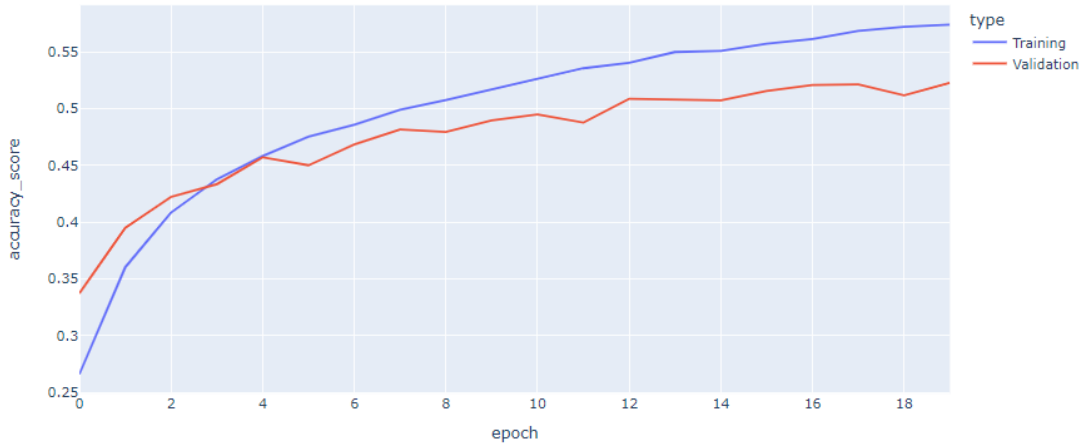
Accuracy Model 1


Accuracy Model 1V2

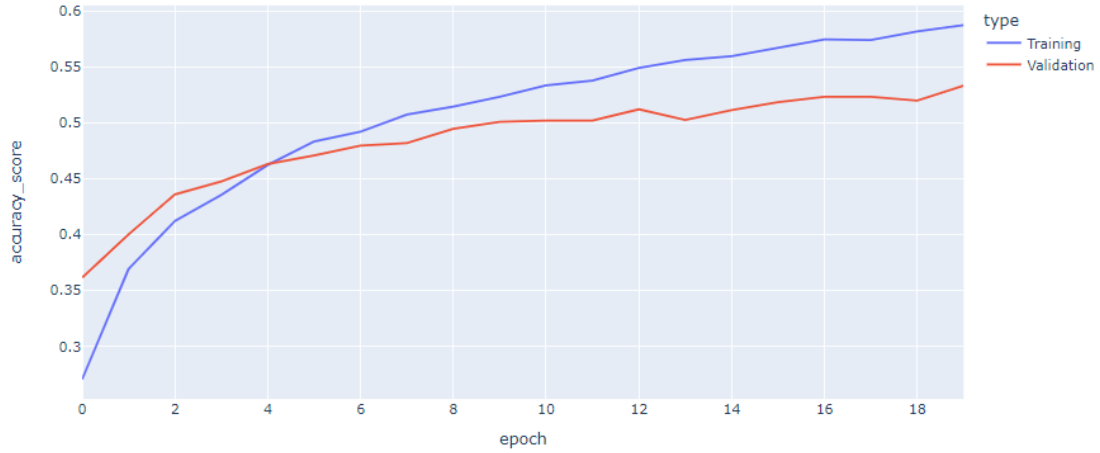

Accuracy Model 1V3



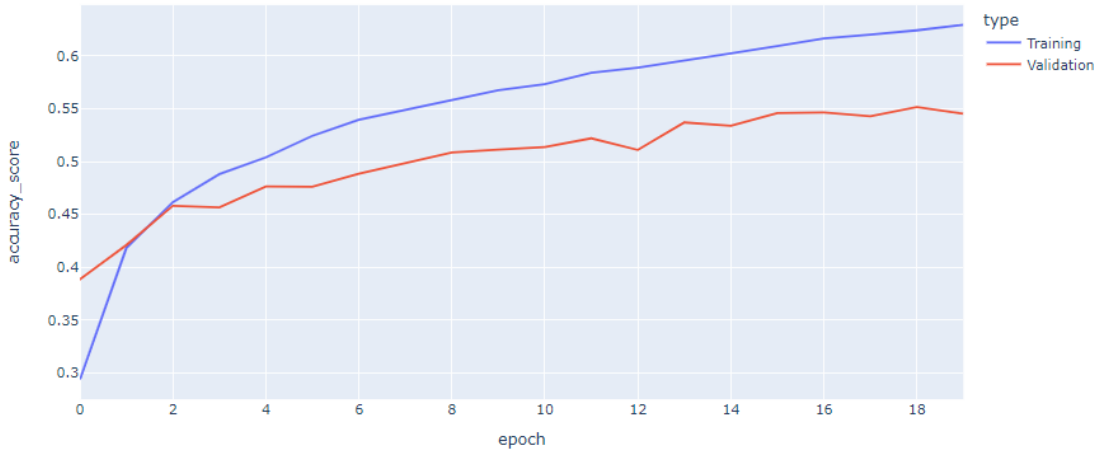
Accuracy Model 2



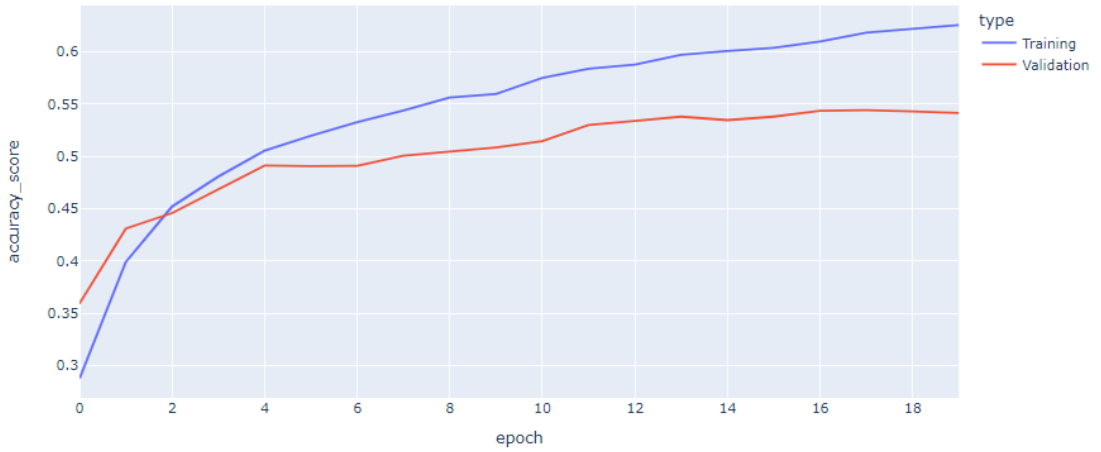

Accuracy Model 2V2



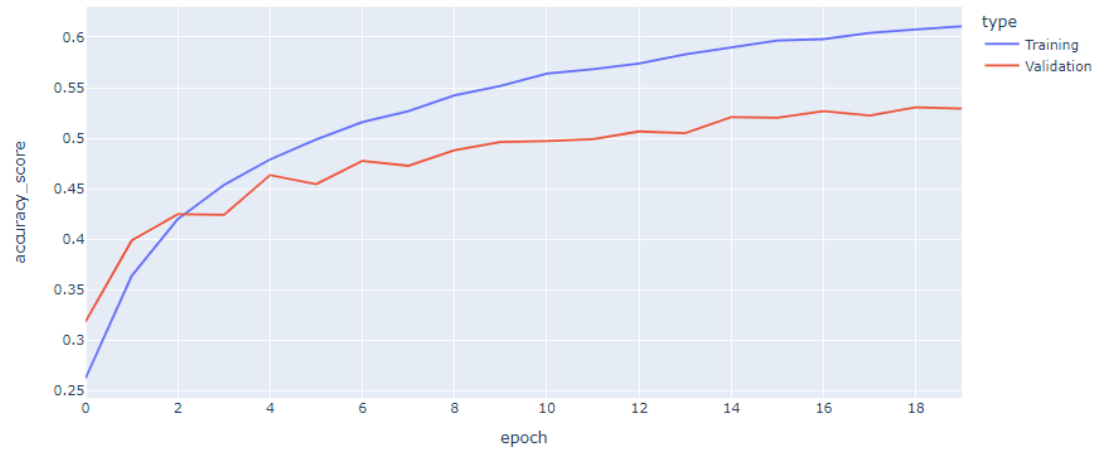
Accuracy Model 3

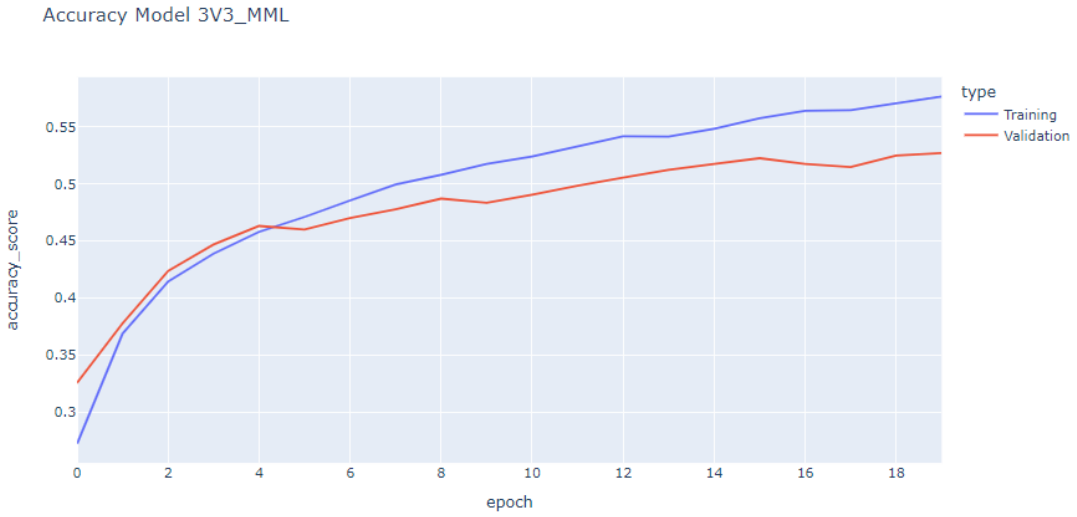



Accuracy Model 3V2

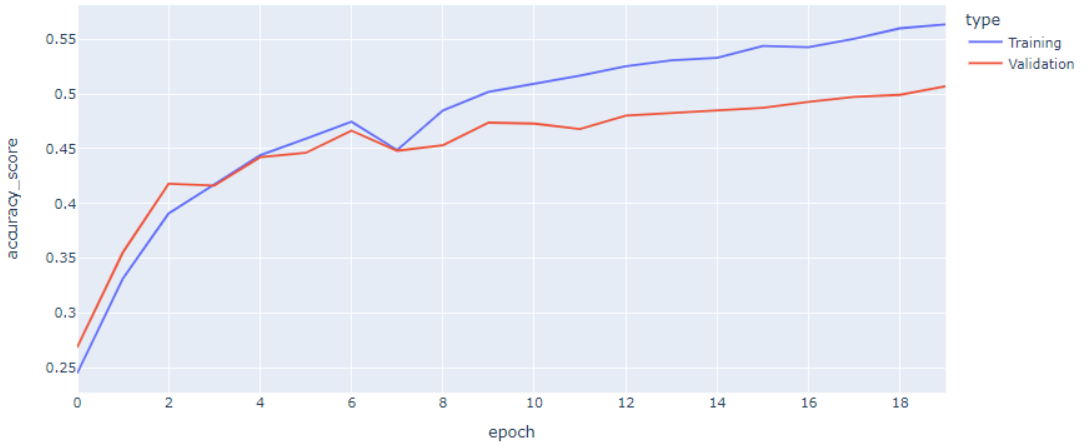



Accuracy Model 3V3

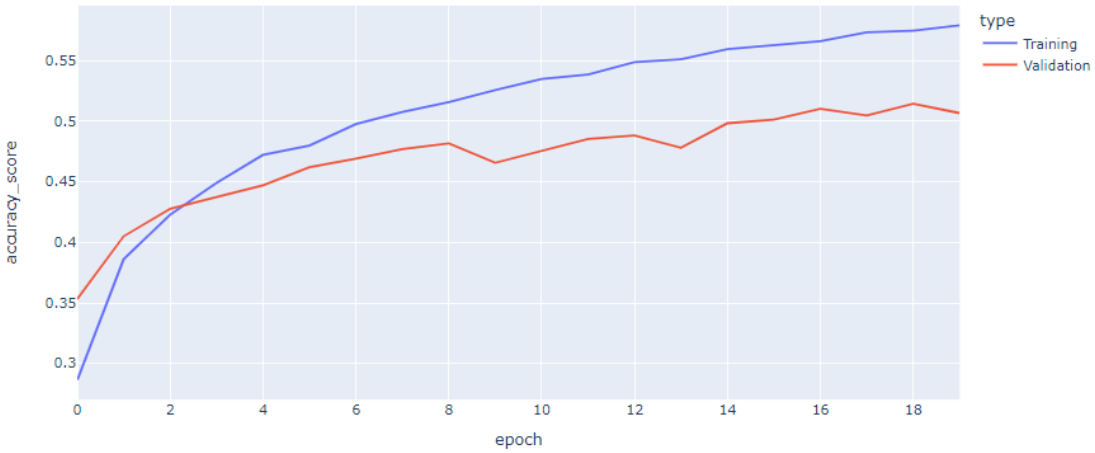



Accuracy Model 3V3_MML
type
Training
Validation
accuracy_score
0.55
0.5
0.45
0.4
0.35
0.3
0
2
4
6
8
10
12
14
16
18
epoch

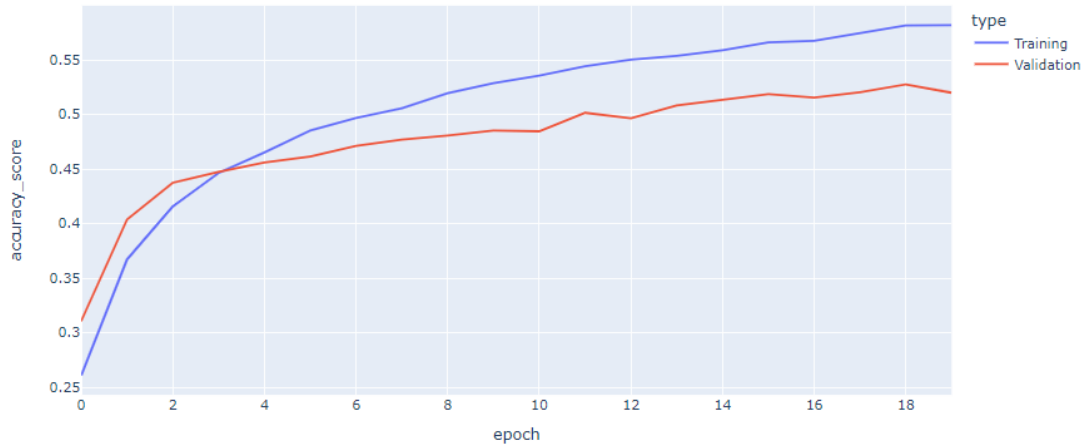
Accuracy Model 3V4



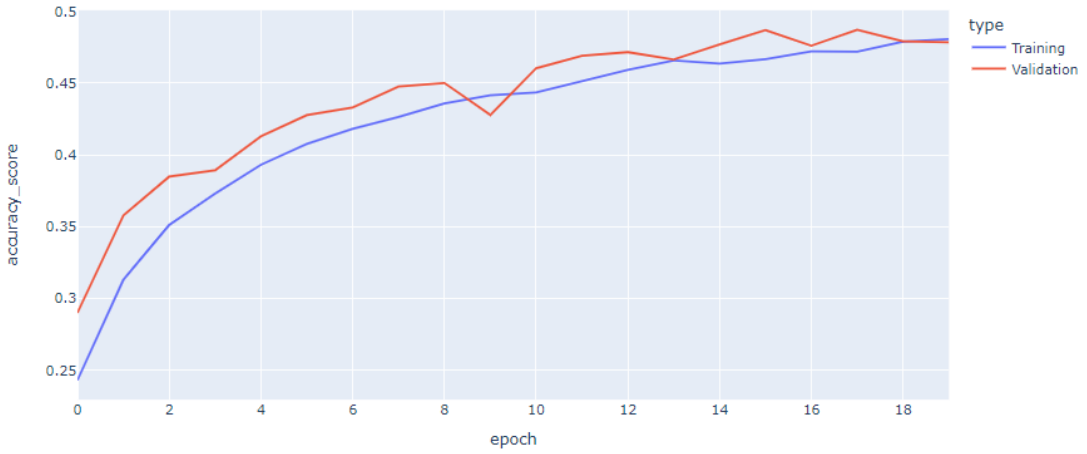

Accuracy Model 4



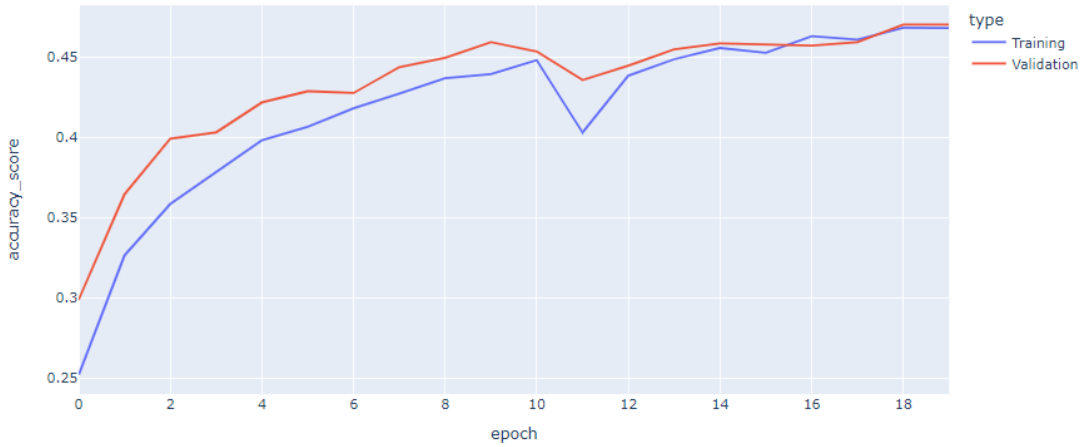

Accuracy Model 4V2



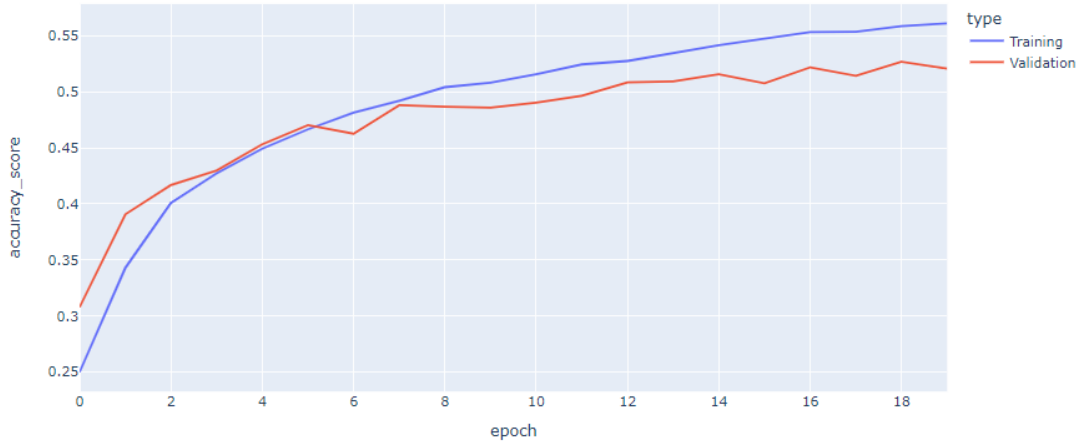
Accuracy Model 5



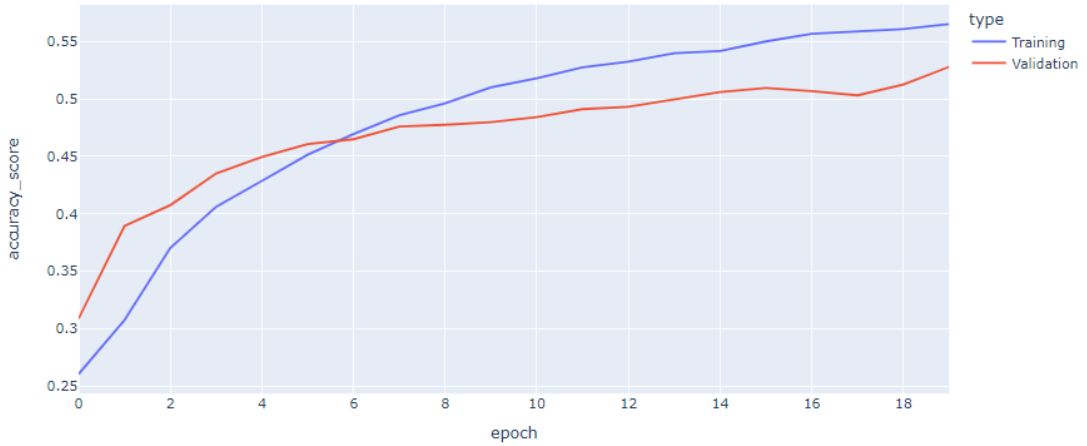

Accuracy Model 5V2



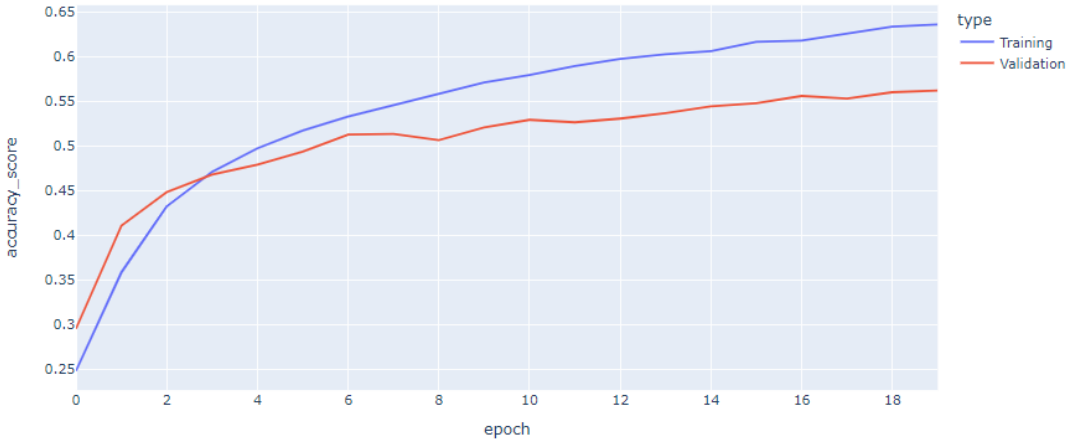

Accuracy Model 6



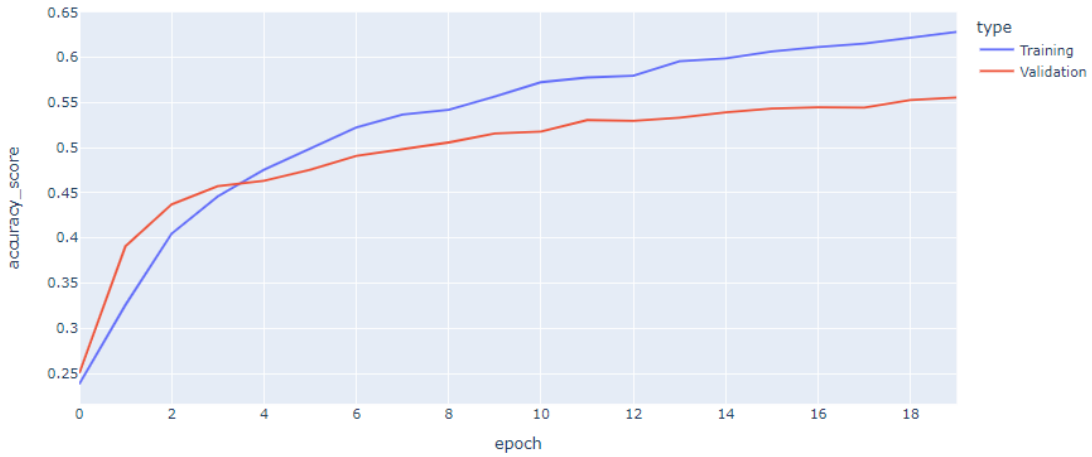
Accuracy Model 6V2



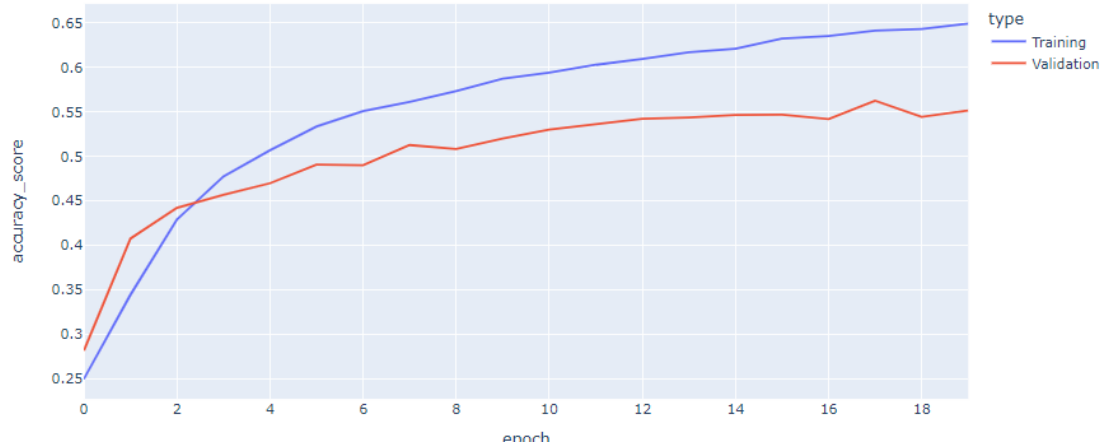

Accuracy Model 7



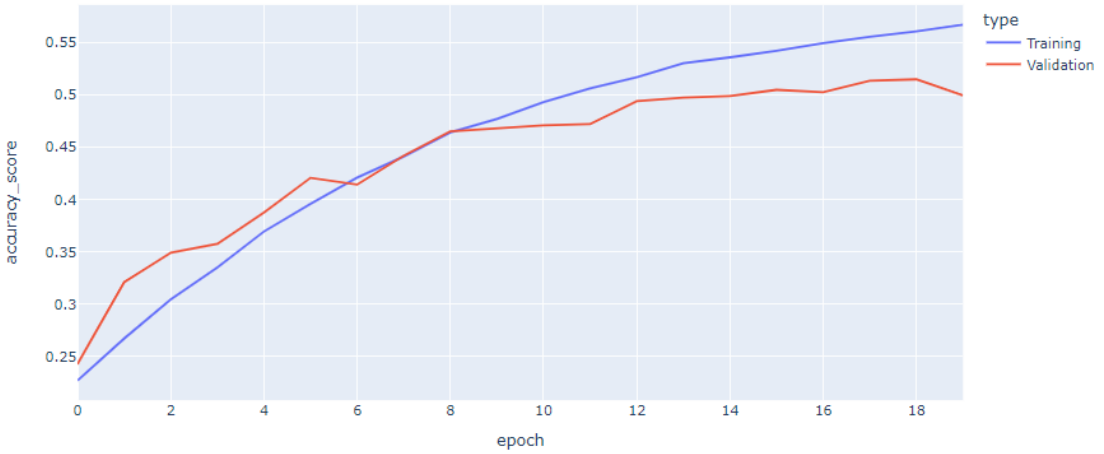

Accuracy Model 7V2



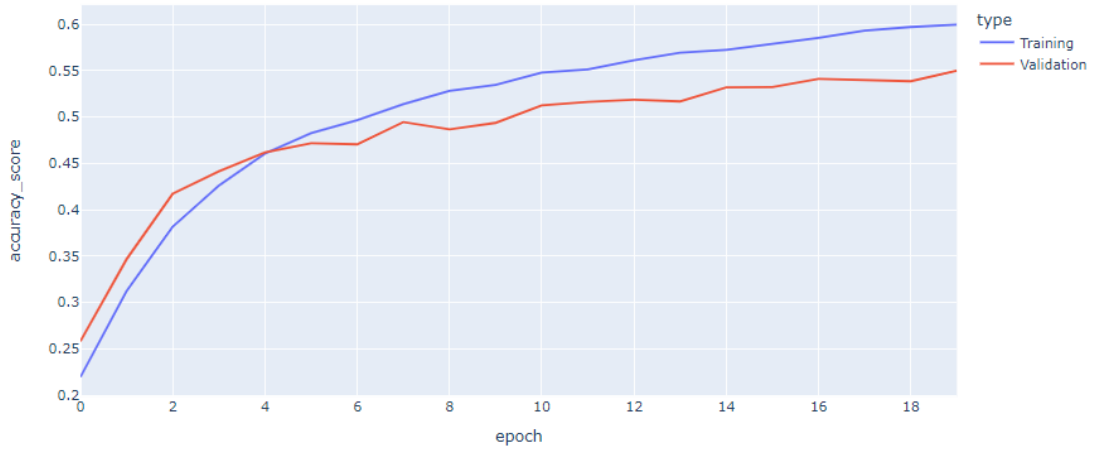
Accuracy Model 7V3




Accuracy Model 7V4

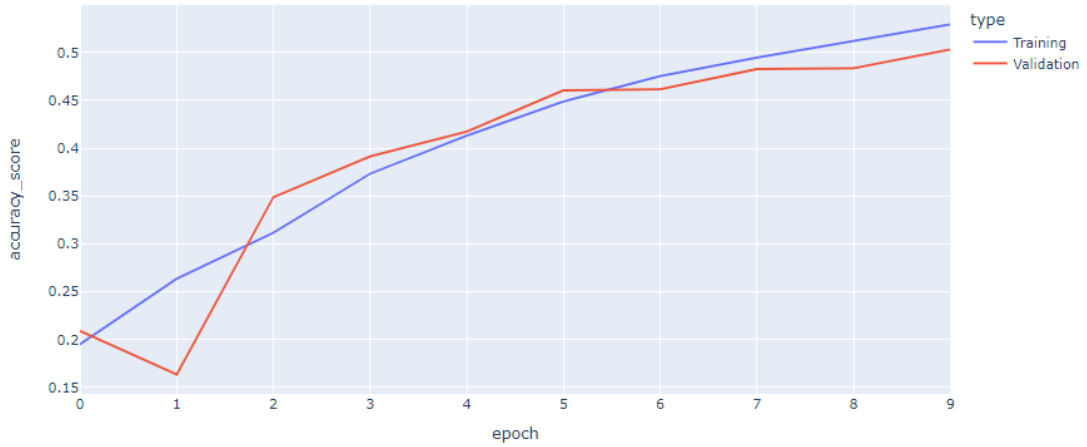



Accuracy Model 8

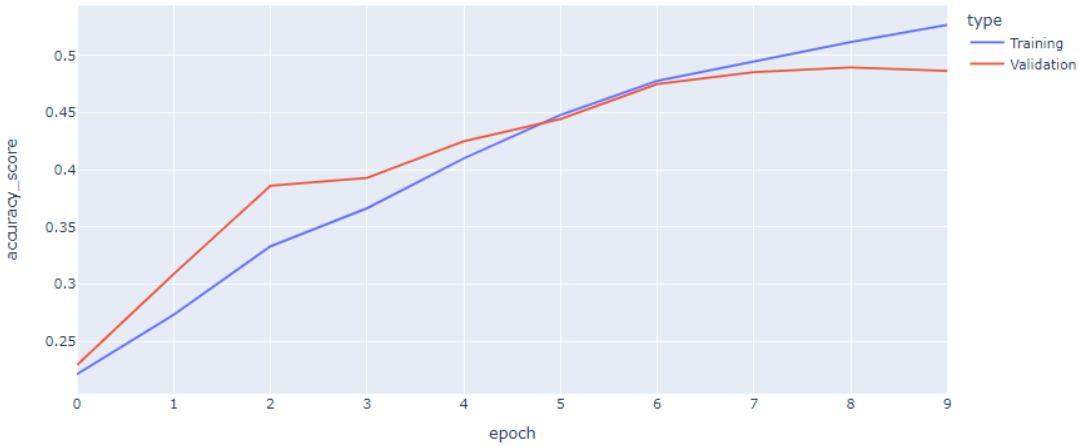


Accuracy Model 8V2


Accuracy Model 9




Accuracy Model 9V2



A lot of conclusions can be drawn from the extensive testing that was performed:

For all models, we can see that there is not much difference between using Adam and RMSProp optimizers, however in most test cases that we have read, the Adam optimizer gives faster results, so it is generally preferred and regarded as the best optimizer for such cases.

Moreover, we can see that the optimum value for learning rate is 1e-4 or 1e-3, as we can see that in every case, lr=1e-2 produces worst results than lr=1e-3 or lr=1e-4.

Comparing Models 3 and 1 with 2 and 4, they have similar architectures but in Models 3 and 1 we have each activation function before each batch normalization, while in Models 4 and 2 we have each activation function after each normalization. We can clearly see that Models 1 and 3 perform better than Models 4 and 2, and so we can draw the conclusion that in such test cases, the batch normalization should come after the activation function. This is compliant with the theory, as we know that the activation function blocks some inputs, which clearly leads to some information being lost.

Regarding max pooling, we can see that Model 3 (with pooling after each layer) has almost the same accuracy as Model 1 (without pooling after each layer), while running much faster due to less parameters. This means that max pooling after each layer seems to be a good tactic when we want to speed up computations without losing accuracy.

Regarding filter sizes and convolution layers we can say the following:

Comparing Model 5 with Models 1 and 7, one of their key differences is the maximum number of channels in them: Models 1 and 7 reach 256 channels while Model 5 reaches 32. Model 5 clearly has worse test metrics than 1 and 7, and so we can draw the conclusion that in order to extract more information about the dataset, the convolution layers should increase their channels as we move forward in the model, with the ideal structure being:

1->32, 32-> 64, 64-> 128, 128 -> 256 channels.

Also, Model 6 reduces its convolution kernel size as we move forward in the model, while Models 1, 7 and 8 increase this kernel size, while having better test metrics. This is in line with the theory of convolution layers, which states that the best method tends to be increasing kernel sizes as we progress through the layers, because in the first layer, we want to capture a lot of details about the images, while in the later stages we are looking for more broad patterns.

Regarding activation functions we can say the following:

Leaky_ReLu with a negative slope of about 0.005-0.01 (like Model 7) seems to be the best choice overall, as it does not allow for the vanishing gradient problem to appear.

Regarding fully connected (linear) layers we can say the following:

Looking at Models 1, 3, 7 against Models 5 and 6, the best practice seems to be having 3 fully connected layers and then a softmax function to the output, with 1 normalization and 1 dropout (0.2) before the final output, which means that FC layers have to decrease output features in multiple stages rather than abruptly, in order to achieve maximum accuracy.

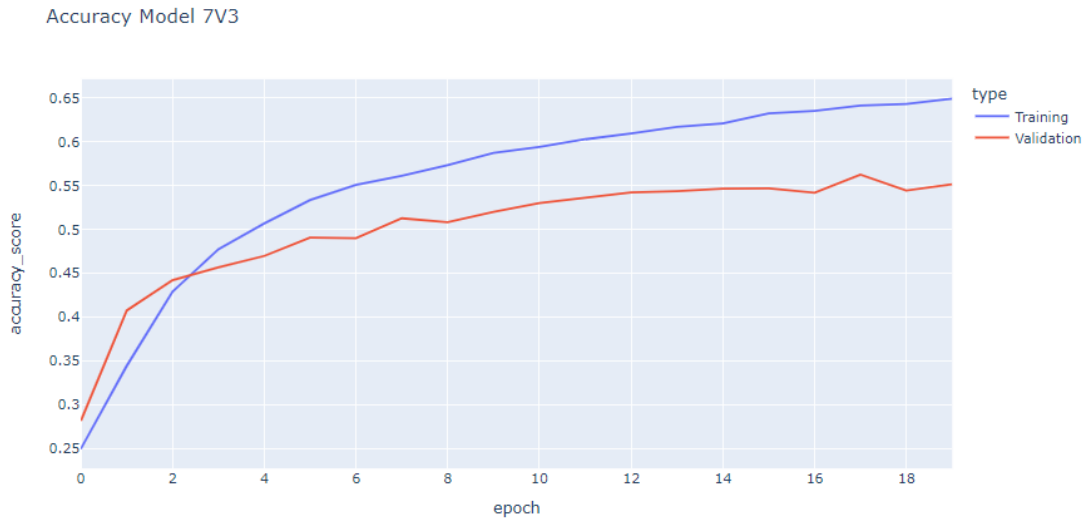
# Step 6 Selected Model Metrics

For all models that were presented, the scores “test accuracy”, “test precision”, “test recall” and “test f1 score” were calculated in order to see which model performs the best for this dataset. From these values, we can see that the overall best performing model is MODEL 7V3:

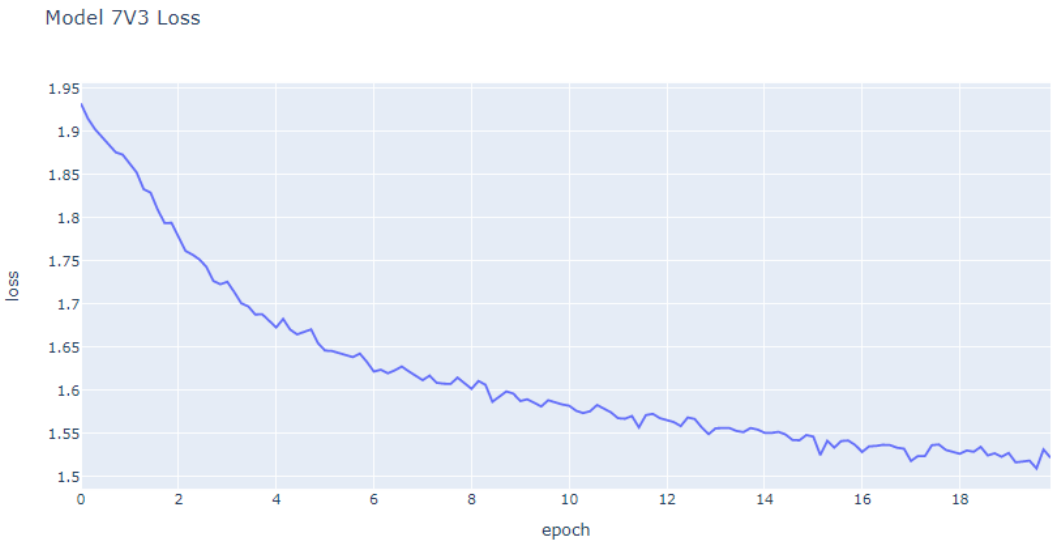
```python
class MODEL7V3(nn.Module):
    def __init__(self):
        super(MODEL7V3, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 5, padding='same')

        self.conv2 = nn.Conv2d(32, 64, 5, padding='same')
        self.batchnorm1 = nn.BatchNorm2d(64)
        self.pool = nn.MaxPool2d(2, 2)


        self.conv3 = nn.Conv2d(64, 128, 7, padding='same')

        self.conv4 = nn.Conv2d(128, 256, 7, padding='same')
        self.batchnorm2 = nn.BatchNorm2d(256)


        self.fc1 = nn.Linear(36864, 128)
        self.batchnorm3 = nn.BatchNorm1d(128)
        self.dropout1 = nn.Dropout(.2)
        self.fc2 = nn.Linear(128, 32)
        self.batchnorm4 = nn.BatchNorm1d(32)
        self.dropout2 = nn.Dropout(.2)
        self.fc3 = nn.Linear(32, 7)


    def forward(self, x, training=False):
        #  // LAYER 1
        x = F.leaky_relu(self.conv1(x), negative_slope=0.01)

        #  // LAYER 2
        x = F.leaky_relu(self.conv2(x), negative_slope=0.01)
        x = self.batchnorm1(x)
        x = self.pool(x)

        #  // LAYER 3
        x = F.leaky_relu(self.conv3(x), negative_slope=0.01)

        #  // LAYER 4
        x = F.leaky_relu(self.conv4(x), negative_slope=0.01)
        x = self.batchnorm2(x)
        x = self.pool(x)

        #  // OUT
        x = torch.flatten(x, 1)
        x = self.batchnorm3(F.relu(self.fc1(x)))
        x = self.dropout1(x)
        x = self.batchnorm4(F.relu(self.fc2(x)))
        x = self.dropout2(x)
        x = F.softmax(self.fc3(x), dim=1)
        return x
```

| Model | Average time/epoch (seconds) | Ran for epochs | Number of Trainable Parameters | Training Accuracy After All epochs (%) | Validation Accuracy After All epochs (%) | Test Accuracy (%) | Test Precision (%) | Test Recall (%) | Test f1 (%) |
|---|---|---|---|---|---|---|---|---|---|
| MODEL7V3 | 100 | 20 | 6,783,559 | 64.94 | 55.18 | 57.15 | 52.55 | 60.46 | 52.43 |

The accuracy plot for this model is:

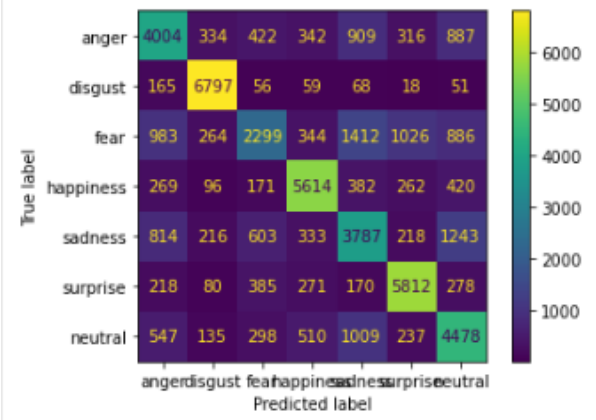


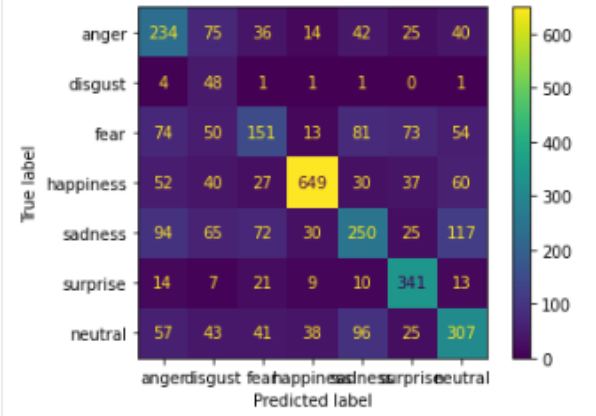

And the Loss plot is:




The confusion matrices for train and validation are:

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x11f91bc7a00>
```

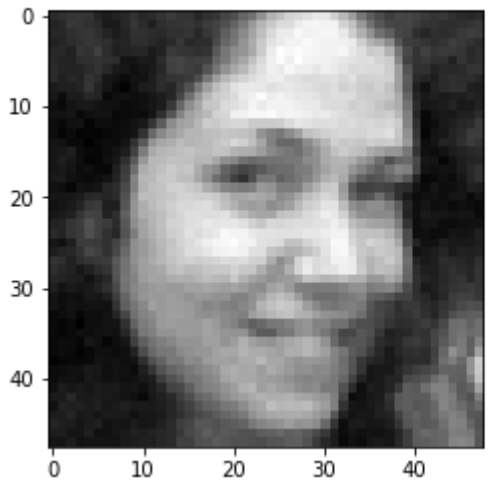


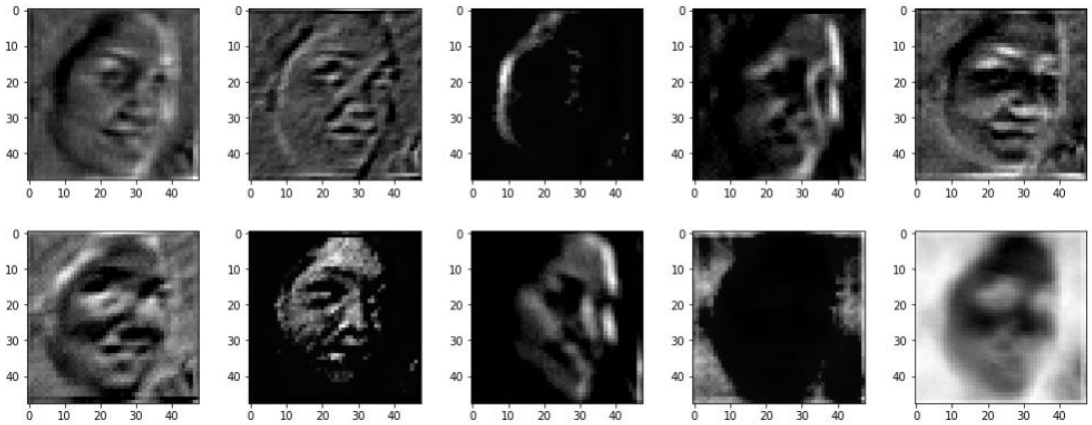

# Step 7 Visualizing Learnt Filters

For the visualization of the learnt filters on every layer, we present the image:

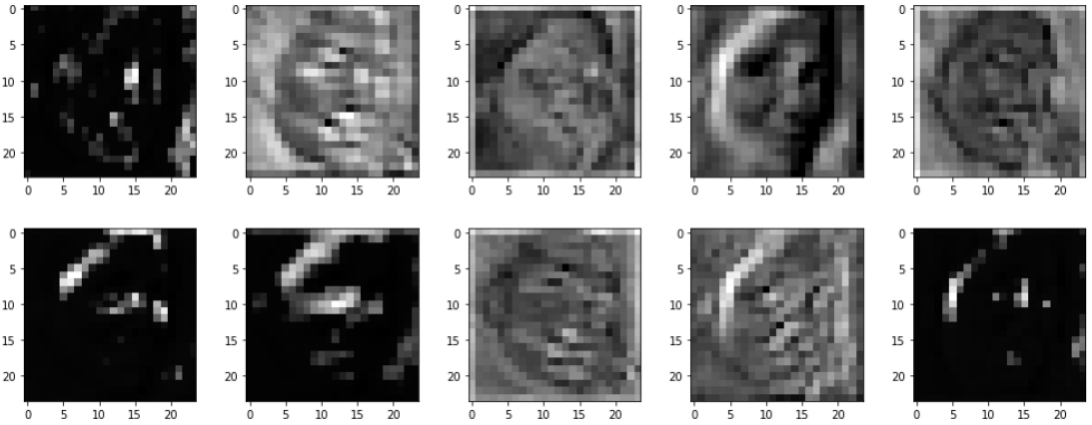
```
<matplotlib.image.AxesImage at 0x11fc0d7d700>
```



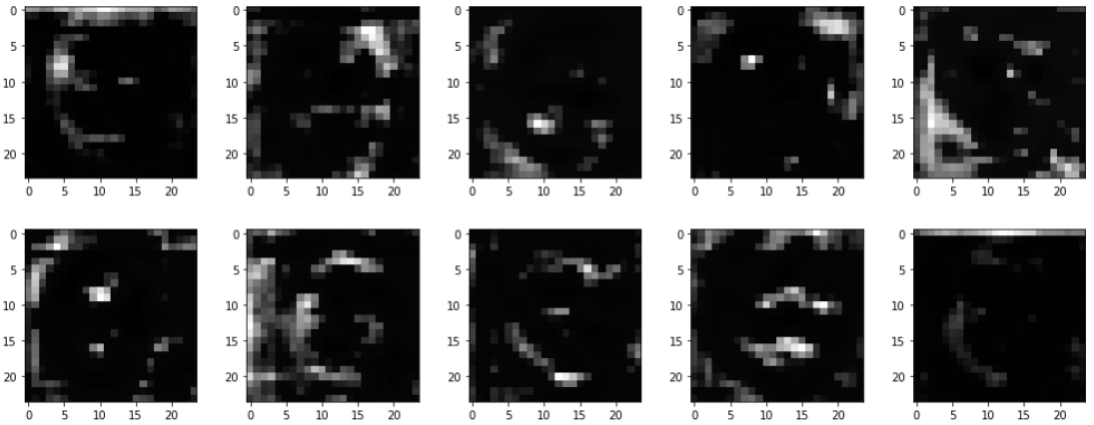
1st layer:



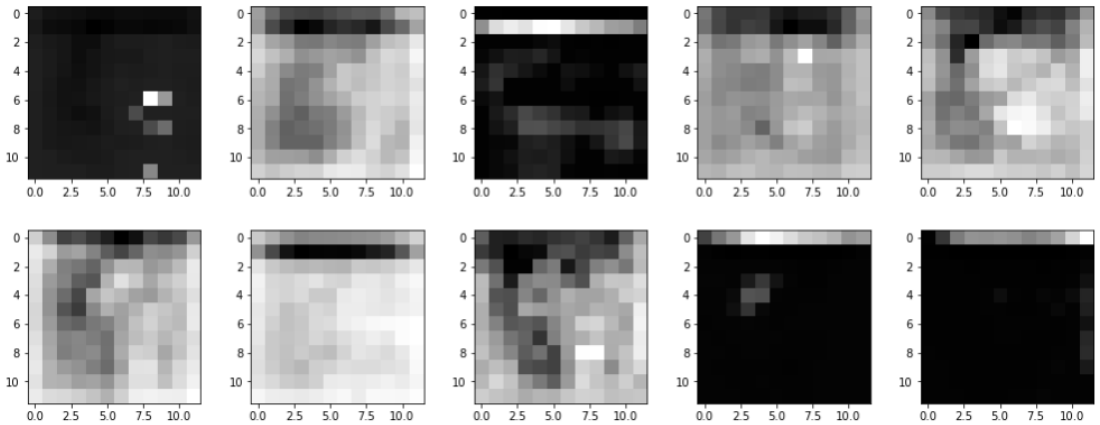
2nd layer:



3rd layer:

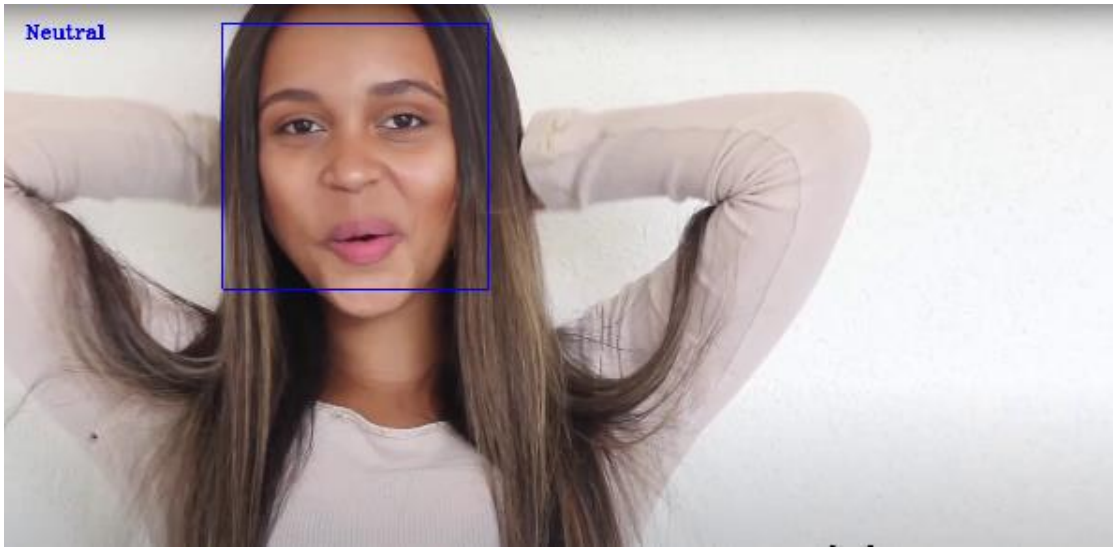


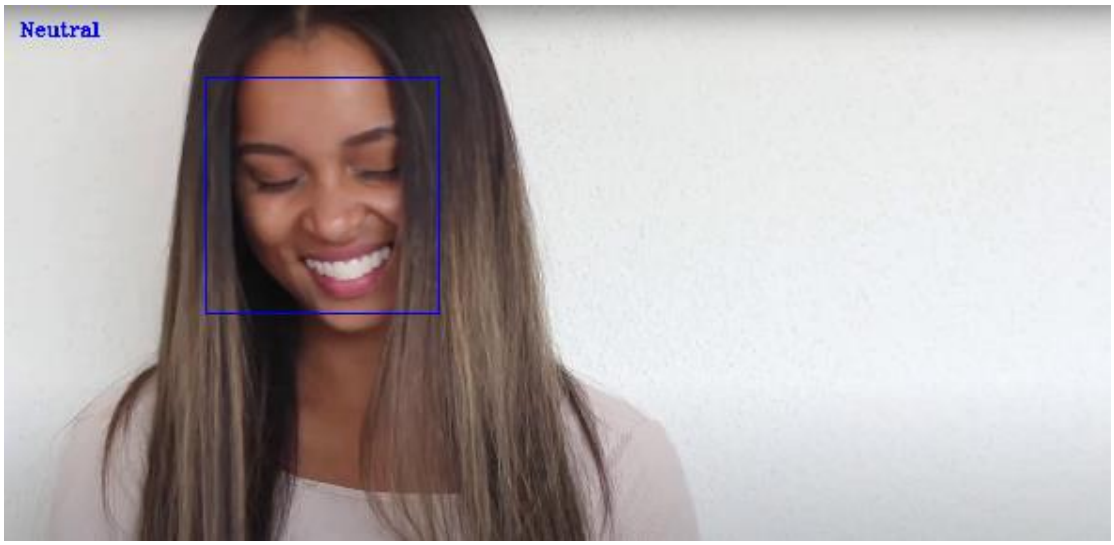
4th layer:

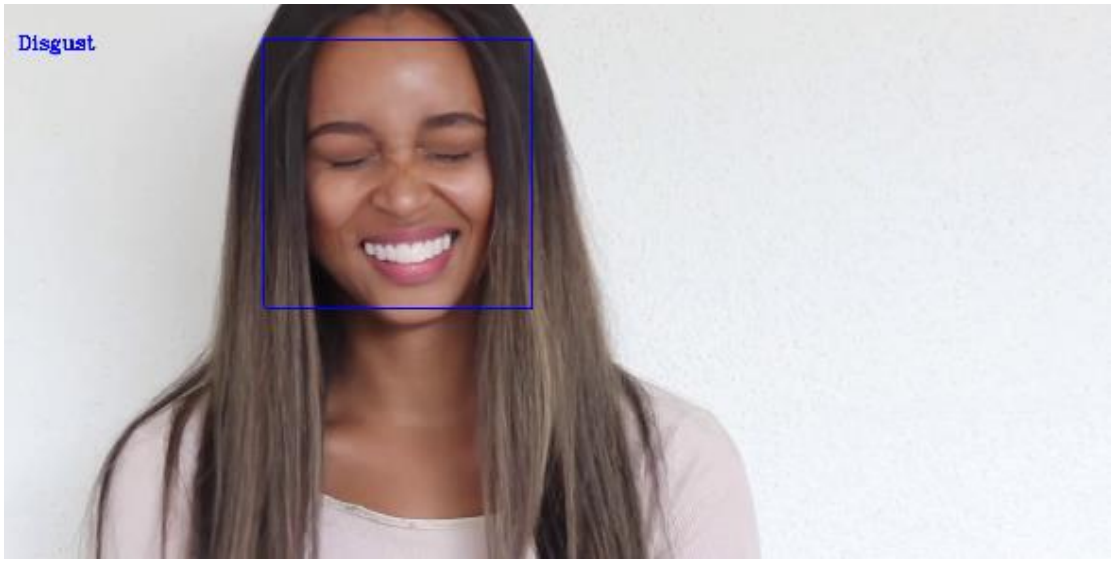


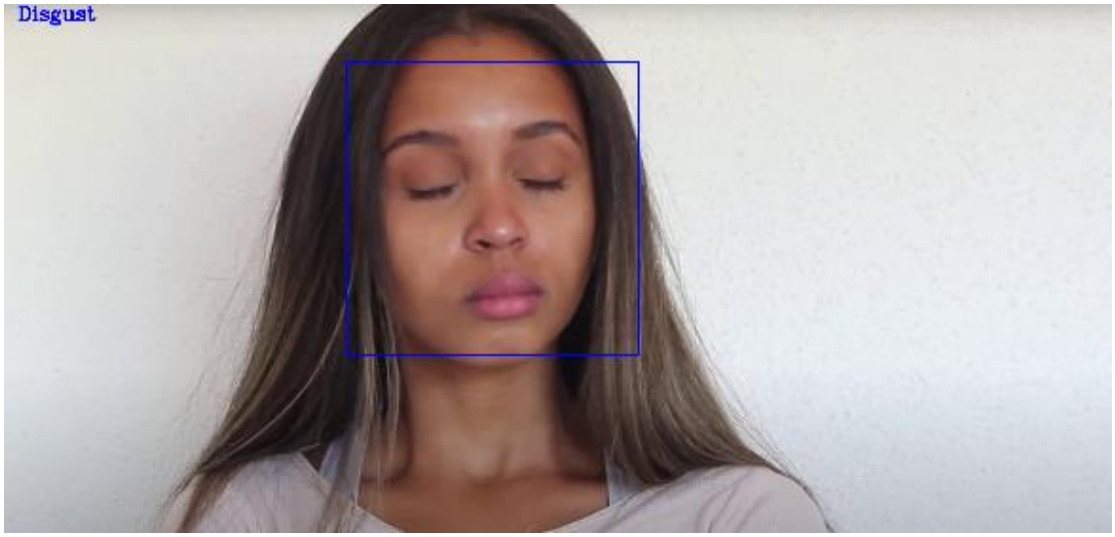
## Step 8 Real Life Videos

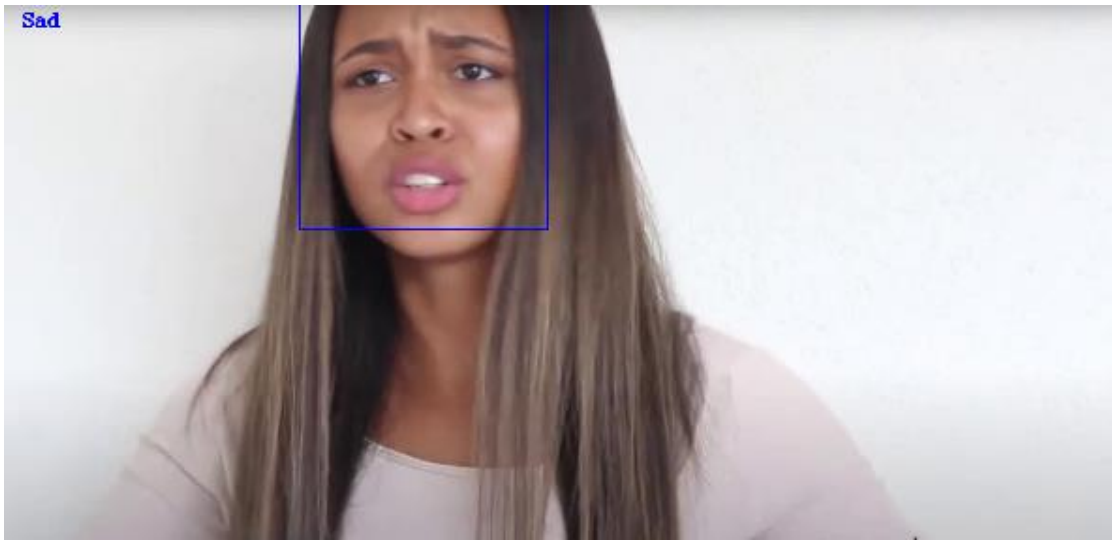
The face detection is computed via the haar cascade method which is built in to opencv. The face is cropped to size (48,48), resized to 4D, converted to Cuda Tensor and fed to the model. From the softmax output, we can see which emotion is predicted. Our model's test accuracy is 57.15% which is somewhat low as an absolute value, however it should be noted that reading through other papers for the same dataset, it seems very difficult for any model to reach above 70% accuracy for FER2013. Considering this, our 57.15% is quite good. However, it is not high enough for almost fault-less predictions and feeding it test videos yields:

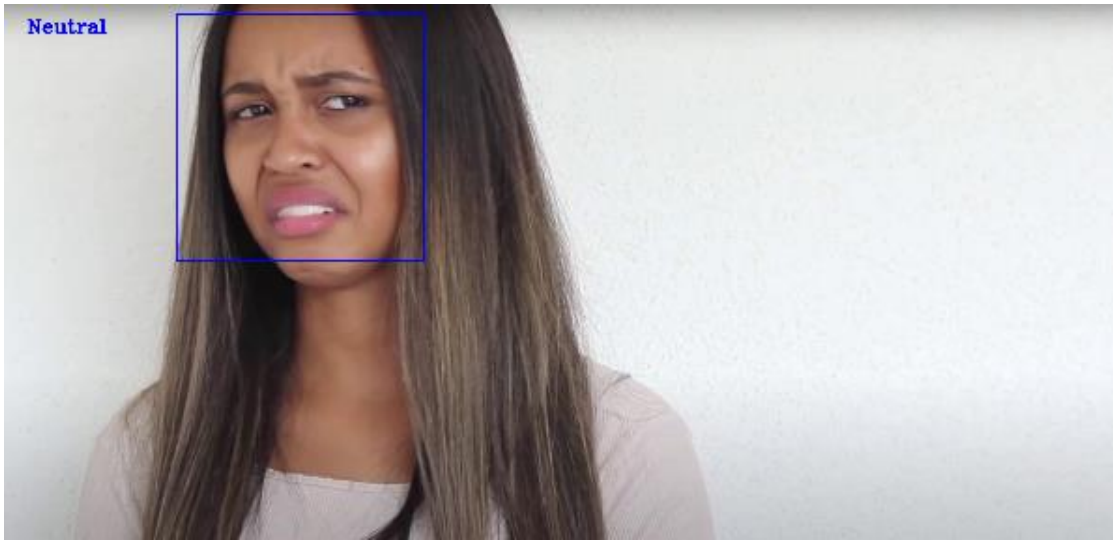

Neutral

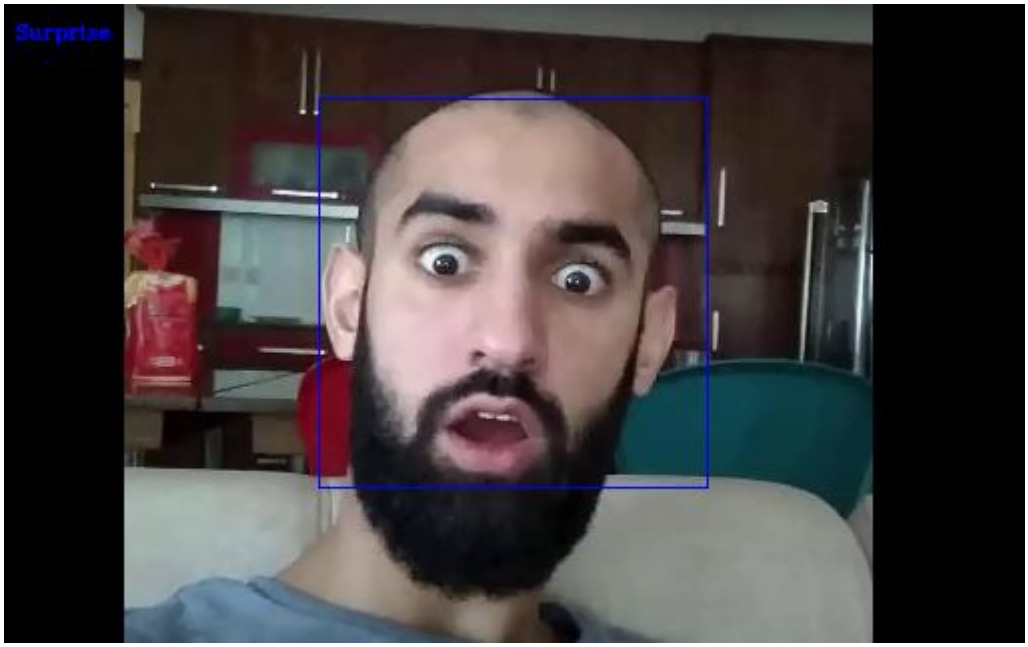

Neutral

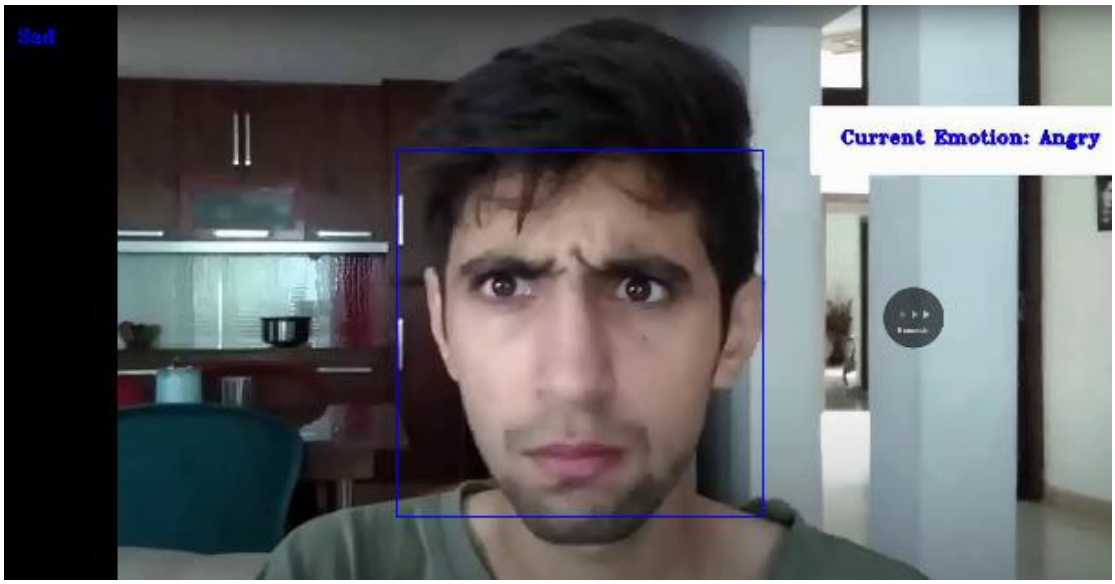

Disgust

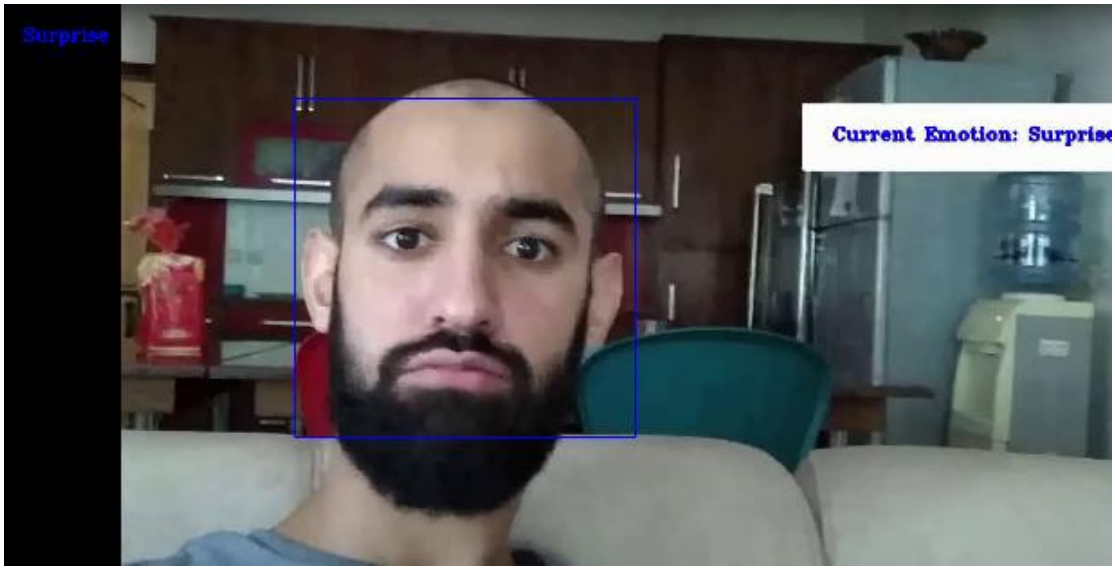

Disgust


Sad


Neutral


Surprise


Sad
Current Emotion: Angry




We can see that our model does not robustly predict the actual emotion of the videos, because its accuracy is somewhat low.

A useful remark about testing this model on a real life video camera:

The model works slightly better if we exaggerate our emotions. This seems to be happening because in the FER2013 dataset, a lot of images are somewhat over-exaggerated representations of images that are difficult to encounter in real life. Because our model is trained on this dataset, it can detect these over-exaggerations with more ease than some somewhat subtle videos that are present on the Internet.

# Step 9 Critical Reflections

From this assignment, we can say that in order to optimize a CNN we must do the following:

- Pick a good dataset, from which our model can learn and reach a very high accuracy. FER2013 allows at most 75% accuracy (as we can see from state of the art models online), and so another model should probably be selected if we want to reach higher accuracy than that.
- Construct the CNN according to our findings: Batch normalization should come after the activation function, max pooling should be implemented somewhat often to reduce running time (reducing number of parameters), kernel sizes should start from 3 and end in 9 or 7 (from layer to layer), and channel sizes should be multiplied by 2 each step (1 to 32 or 64, to 128, to 256, to 512). This helps extract every possible bit of information from the input image. In addition, the convolution layers should be more than 2 (probably 3 or 4 if we want a somewhat fast training CNN, or more if we want a more accurate but slower training one).

- Fully connected layers should be higher than 2 (3 or 4 seem to be optimal). Of course, if we have a very high amount of convolutional layers, the number of FC layers has to increase as well.
- The activation function should be leaky_relu so as to avoid the vanishing gradient problem.
- We should avoid using dropout after each convolution layer, but rather use it every 2 or 3 layers.
- Adam should be the selected optimizer, and cross entropy loss should be the loss function.