	Silesian University of Technology **Biologically inspired artificial intelligence**		
Year	**Type of studies: SSI/NSI/NSM**	**Group**	**Section**
2022/2023	**SSI**	**GKiO**	**1**
Supervisor:	**dr. inż. Grzegorz Baron**	**Classes:**	
Section members:	**Adrian Zaręba**	**Tuesday 9:45-11:15**	
Section emails:	adrizar364@student.polsl.pl		

Report

Topic:

Hand gesture classification

Table of Contents

Project Topic:

The topic of this project is to create and train a neural network capable of recognizing different hand gestures presented on images.
As hand gesture counts e.g. “Ok” sign, “like”, “clenched fist”.

Analysis:

Possible approaches:

There were two fitting approaches to this issue.

One was to treat the topic as an image classification problem.
It usually involves large dataset and some kind of neural network, most likely convolutional neural network in order to train model properly.

Second approach was to use an pose estimation technique. It would detect and mark skeletal features of a hand (like fingers, joints) on a picture.
Created landmarks would allow for easy determination of the gesture.
This technique is often used combined with live videos.
An example of usage of this technology is Goggle’s Mediapipe.

Let’s consider all advantages and disadvantages of mentioned approaches.

Advantages of the first mentioned (Image classification)

- Continuous learning on large datasets allows to improve model’s accuracy
- By providing right data, final model might be able to recognize gestures even in distorted, twisted and rotated images.
- If CNN is in use, it is an effective way of learning of pattern recognition

Disadvantages:

- It would most likely preform much worse on live videos, dynamic motion that blurs the pattern and real time needs would render this approach not useful.
- It is not guaranteed that trained model would focus on right patterns and instead of “looking” at hands it would focus on e.g. background colour.

Pros of latter approach (Pose estimation):

- Possible better performance with real time videos
- Easier expansion of the program with different features as landmarks are more universal data than classes on IC approach.
- Reduced complexity of the recognition task as opposed to processing raw image data as in IC.

Cons:

- Much more difficult implementation
- Need for two solutions: detection of the landmarks and mapping landmarks accurately to gestures.

Conclusion:

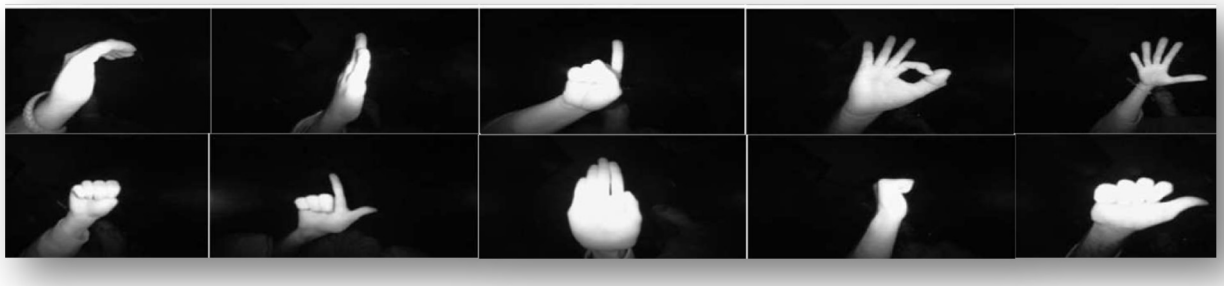
Image classification has been selected as primary approach to the problem.
It is much commonly used solution for problems such as this topic.
It was also expected to yield satisfying results with less mundane work.

Datasets:

As a source of datasets the Kaggle platform was selected as it was highly recommended and reliable.

Hand Gesture Recognition Database
https://www.kaggle.com/datasets/gti-upm/leapgestrecog

Dataset composed of a set of near infrared images acquired by the Leap Motion sensor. It consists of 10 classes (hand gestures).
Gestures were performed by 10 subjects .
Data is split into folders with corresponding names. There is total of 20000 images in this dataset.



Picture 1 Hand Gesture Recognition Database

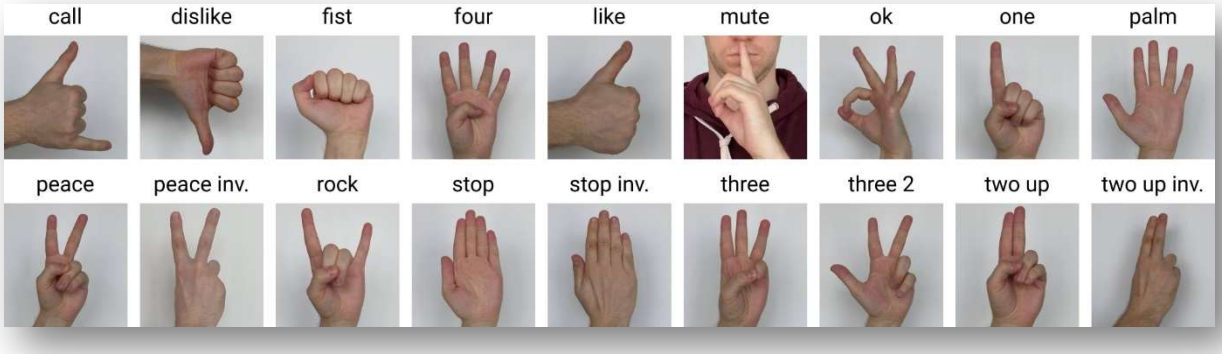
Hand Gesture Recognition
https://www.kaggle.com/datasets/roobansappani/hand-gesture-recognition

Dataset created by subtracting background form the images and discoloration to black and white images.
It consists of 10 classes, 500 images each (5000 images in total).

Picture 2 Hand Gesture Recognition

HaGRID - Hand Gesture Recognition Image Dataset
https://www.kaggle.com/datasets/kapitanov/hagrid

This dataset is a collection of images gathered remotely from volunteers (34.730 people).
Its full size is 716GB and contains total of 552992 HQ images.
There is 18 classes (gestures) in total. Each image is named with subject id.
All images are in same folder however their names are bound to data in json file which hold all necessary information about image class and hand/hands position on the image.
The subjects are people from 18 to 65 years old. The dataset was collected mainly indoors with considerable variation in lighting, including artificial and natural light.
Besides, the dataset includes images taken in extreme conditions such as facing and backing to a window.
Also, the subjects had to show gestures at a distance of 0.5 to 4 meters from the camera.
Kaggle provides also a smaller subset of dataset above. Its size is reduced to 2.5 GB.



Picture 3 HaGRID - Hand Gesture Recognition Image Dataset

Cropped_Hand_Gestures_From_HAGRID_Dataset
https://www.kaggle.com/datasets/ruckdent/cropped-hand-gestures-from-hagrid-dataset

It is a subset of earlier mentioned HaGrid dataset. Some images were taken from original dataset and hands were cropped out (based on landmarks in json file) of these images, divided and saved in corresponding folders.
This dataset consists of ~10.000 images and has 18 classes as well.

Picture 4 Cropped_Hand_Gestures_From_HAGRID_Dataset

Selected Dataset:

As first two datasets: **Hand Gesture Recognition Database** and **Hand Gesture Recognition** were really interesting, considering techniques on how they were prepared, they were also quite narrow as only black & white or grayscale images were presented.
Small variety of pictures even with reasonable numbers disqualified them.

Finally the chosen dataset was **Cropped_Hand_Gestures_From_HAGRID_Dataset** as it is a subset of **HaGRID - Hand Gesture Recognition Image Dataset** which is reasonable too big for the scale of this project,
yet **Cropped_Hand_Gestures_From_HAGRID_Dataset** still has most of the advantages of original **HaGrid** like variety of images and environments.
It also has its data cleanly prepared (focus on hands) which came in handy later on.

Tools & Frameworks & Libraries:

PyTorch
It is a popular deep learning framework that is extensively used and recognized for its dynamic computational graph and intuitive interface.
It offers robust support for CNNs and enables efficient development and training of models. PyTorch also provides GPU acceleration and integration with Python

TensorFlow
It is an extensively utilized deep learning framework known for its popularity in training and deploying CNN models. It presents an extensive range of tools and application programming interfaces for constructing and training neural networks, specifically CNN structures. TensorFlow facilitates GPU acceleration to enhance training speed and enables model deployment on multiple platforms.

Colab
It is a cloud-based platform developed by Google. It provides a free environment to execute Python code. It offers the convenience of writing and running Python code directly in a web browser, eliminating the need for any setup or installation. Google Colab supports free GPU/TPU.

Keras
It is a deep learning library that operates at a high level and can be utilized alongside TensorFlow or other backend frameworks. It offers a user-friendly interface specifically designed for the creation and training of CNN models.

Jupiter Notebook
It is an open-source web application that allows users to create and share documents containing live code, equations, visualizations, and narrative text.

Visual Studio Code
It is a free source code editor developed by Microsoft. It provides a versatile and feature-rich environment for software development across multiple programming languages. It supports various programming languages and offers features like code highlighting, debugging, version control integration, and intelligent code completion.

Selected tools & frameworks & libraries:

VS Code – it was a personal preference as it is a well-known environment for the section.
It allows for classical style of writing code and programming.

PyTorch – as many people tend to choose Tensorflow over Pytorch, the latter offers slightly more control over your code, hiding less behind layers of abstraction.

- Library matplotlib - used for creating high-quality visualizations and plots.
- Library numpy – basic library for operations on matrixes and arithmetic
- Library sklearn – for easy and convenient implementation of confusion matrix
- Library seaborn - data visualization library built on top of Matplotlib.
- Library pandas - used for data manipulation and analysis
- Library pillow - library used for working with images.

External specification

Usage

Run main.py file that is placed within working directory of the project.
Use Python 3.7 or newer.

While using the program it is possible to change some of the parameters of the project,
either by intervention in the code itself or by passing arguments in command line (achieved by using flags).

Parameters that might be safely changed by edition of the code:

- Boolean *training* – specifies if program will train or test model. Set true to train.
- Model *name* – a parameter of LoadModel function call, change when in test mode to test different saved model.
- *BATCH_SIZE* – size of batch, default: 64. Integers only.
- *NUM_EPOCHS* – number of epochs, default: 25, Integers only.
- *specsModelInfo* – a char string that will be added to the end of a name of saved model in training model. It is used to pass additional information about the model.

Parameters in code that shouldn't be changed:

- Anything outside of main.py unless different behaviour of the program is wanted.
- *SEED* and *SPLIT* – it would be better to leave it as it is for consistence of results.
- *IMAGE_SIZE* – it would require more changes in the model itself.

Command line flags:

- -tr True/False – edits training parameter, Boolean value
- -bch [INT] – edits BATCH_SIZE parameter, positive integer
- -ep [INT] – edits NUM_EPOCHS parameter, positive integer
- -inf [TEXT] – edits specsModelInfo parameter, any string
- -ld [TEXT] - edits Model name parameter, provide name of saved model

Output

While training models it is possible to save trained model.
It is saved in working-dir/trainedFinals. After save the program ends.

While testing a model, there will pop out an message box containing two graphs of loss function and accuracy function, graph can be saved by using button on the bottom.
Generated heatmaps are automatically saved in working directory (most often – overridden).

Input

While testing model it is possible to change which of the saved models will be loaded. Replace the name of the model within the main function in call of LoadModel function.
For learning, there is specified path for dataset directory that shouldn't be changed unless the dataset is moved. Don't change catalogue structure of the dataset. Changing names of catalogues is allowed, renaming files, deleting and adding new images is also allowed. Don't put any other files that jpg and png inside dataset as it may result in unexpected behaviour.

Internal Specification

Classes:

ConfusionMatrixGenerator – class responsible for testing created model and getting its output solely to generate and save as png file a confusion matrix. It has one static method for this purpose.

```
class ConfusionMatrixGenerator:

    @staticmethod
    def generateConfMatrix(net, testSet, filename:str):
```

Picture 5 ConfusionMatrixGenerator

DatasetFromFolderCustom – a descendant class of Dataset form PyTorch lib. Loads, shuffles and splits dataset. Its most important method is _getitem_.
It is used to get data from the dataset anytime during training or testing as it allows for iteration.

```
#Override
def __getitem__(self, index: int) -> Tuple[torch.Tensor, int]:
    img = self.load_image(index)
    className  = self.paths[index].parent.name
    classIndex = self.classToIndex[className]

    if self.transform:
        return self.transform(img), classIndex
    else:
        return img, classIndex
```

Picture 6 DatasetFromFolderCustom

CustomTinyVGG - a descendant class of PyTorch Module. It resembles a TinyVGG architecture as it has 4 convolutional layers. It has been modified multiple times and now is quite different from initial structure, at least in current state.

```
class CustomTinyVGG(nn.Module):
    def __init__(self, input_shape: int, hidden_units: int, output_shape: int) -> None:
        super().__init__()

        self.convBlock_1 = nn.Sequential(
            nn.Conv2d(in_channels=input_shape,
                      out_channels=hidden_units,
                      kernel_size=6,
                      stride=1,
                      padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=hidden_units,
                      out_channels=hidden_units,
                      kernel_size=6,
                      stride=1,
                      padding=1),
            nn.Dropout(0.9),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2,
                         stride=2)
        )
        self.convBlock_2 = nn.Sequential(
            nn.Conv2d(hidden_units, hidden_units, kernel_size=6, padding=1),
            nn.ReLU(),
            nn.Conv2d(hidden_units, hidden_units, kernel_size=6, padding=1),
            nn.Dropout(0.9),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(in_features=pow(32,2)*hidden_units, out_features=output_shape)
        )

    def forward(self, mod: torch.Tensor):
        mod = self.convBlock_1(mod)
        mod = self.convBlock_2(mod)
        mod = self.classifier(mod)
        return mod
```

Picture 7 CustomTinyVGG

RasNet - a descendant class of PyTorch Module. It is strongly based on RasNet18 architecture.

```
class ResNet(nn.Module):
    def __init__(
        self,
        img_channels: int,
        num_layers: int,
        block: Type[BasicBlock],
        num_classes: int  = 18
    ) -> None:
        super(ResNet, self).__init__()
        if num_layers == 18:
            layers = [2, 2, 2, 2]
            self.expansion = 1

        self.in_channels = 64
        self.conv1 = nn.Conv2d(
            in_channels=img_channels,
            out_channels=self.in_channels,
            kernel_size=7,
            stride=2,
            padding=3,
            bias=False
        )
        self.bn1 = nn.BatchNorm2d(self.in_channels)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self.makeLayer(block, 64, layers[0])
        self.layer2 = self.makeLayer(block, 128, layers[1], stride=2)
        self.layer3 = self.makeLayer(block, 256, layers[2], stride=2)
        self.layer4 = self.makeLayer(block, 512, layers[3], stride=2)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512*self.expansion, num_classes)
    def makeLayer(···

    def forward(self, x: torch.Tensor) -> torch.Tensor:···
```

Picture 8 RasNet

Free functions:

testModelVisually(testDataset, model, imageSize, contrFun, contrast, numOfImgs) – function allows user to see model in action. Displays provided number of images from the dataset with labels that represent best guesses of the model together with percentage of confidence.

grepClasses(directory) – Returns classes for the classification based on folder names of the dataset

trainStep(model, dataLoader, lossFun, optimizer, device) – trains model for one epoch, calculates loss and accuracy.

testStep(model, dataloader, lossFun, device) –calculates loss and accuracy of the model on test dataset.

trainAndStat(model, trainDataLoader, testDataLoader, optimizer, lossFun, epochs, device) – combines trainStep and testStep() function and trains model through all epochs.

LoadModel(path, filename) – self explanatory

SaveModel(model, path, filename) – self explanatory

plotLoss(results, epochs) – creates a plot of epochs to accuracy and a plot of epochs to loss

Main script:

Script that controls flow of the program and accepts user input as described in external specification. Has two main separate tasks. Train or test CNN models.

Always:

Initializes universal variables.
Creates transform objects for datasets (image processing pipelines).

Training:

Loads dataset divided into train and test. Initializes a model.
Sets a loss function and optimizer. Trains model with trainAndStat function.
Prints results and time during that time.
If no errors occurred, model is being saved.

Test:

Loads test part of the dataset. Loads saved model. Runs testModelVisually function and generateConfMatrix method of ConfusionMatrixGenerator class.

Experiments:

Experimenting with TinyVGG architecture:

The main focus of the experiments was to discover best parameters for a single model.
It was achieved by making slight variations in models structure and parameters. Usually one at the time in order to see its impact on the network's learning clearly.
The overall configuration of the model was shown in internal specification section. It was a base for all further experiments.
The images were initially loaded at a size of 128x128, however different resolutions were tested as well.

Table 1 Experiments with TinyVGG

3	Number of conv layers	4	4	4	4	4	4	4	6	4	4	4	2	4	4	4
4	Padding	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1
5	Dropout	-	-	-	-	-	1x(0.5)	1x(0.5)	1x(0.5)	2x(0.7)	2x(0.8)	2x(0.8)	2x(0.8)	4x(0.6)	4x(0.8)	4x(0.6)
6	Batch	32	32	32	32	62	32	32	32	32	32	32	32	32	32	32
7	Epochs	10	10	10	30	80	10	10	10	10	10	15	15	15	15	100
8	Pre-processing	resize	resize	resize	resize	resize	resize	resize	resize	resize	resize	resize	resize	resize	resize	resize
9	Activation function	ReLU	ReLU	ReLU	ReLU	ReLU	ReLU	ReLU	ReLU	ReLU	ReLU	ReLU	ReLU	ReLU	SeLU	ReLU
10	Image size	128x128	128x128	128x128	128x128	128x128	128x128	256x256	256x256	128x128	128x128	128x128	128x128	128x128	128x128	128x128
11	Learning rate	Default	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.001	0.001	0.0001	0.0001	0.0001	0.0001	0.0001
12	Optimizer	Adam	Adam	Adam	SWG	Adam	Adam	Adam	Adam	Adam	Adam	Adam	Adam	Adam	Adam	Adam
13	Weight loss	-	-	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.005	0.005	0.005	0.005	0.005

- 1
  First try to train the model, default parameters
  Result: Overall performance of the model was great regardless of most important thing which was test accuracy as it was lower than three quoters.
- 2
  Kernel size of convolution layers was increased to “catch” more information form pictures. Additionally, small padding was added and learning rate was explicitly given.
  Result: Compared to previous try, all measurements improved slightly.

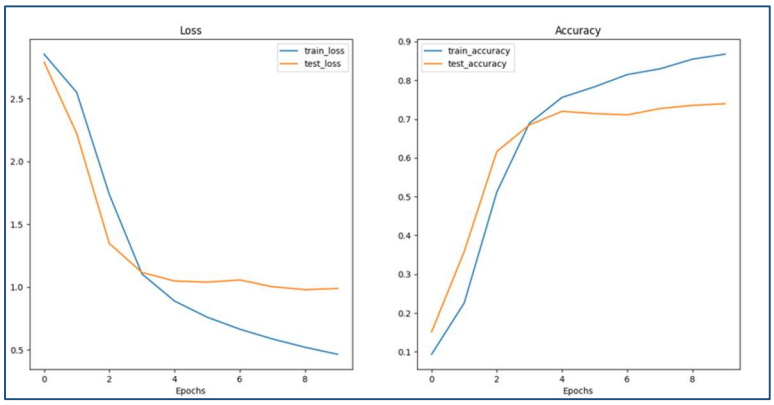


Figure 1 TinyVGG experiment 2

- 3
  Weight loss for optimizer function was introduced.
  Result: Slight improvements on the field of the train and test loss.

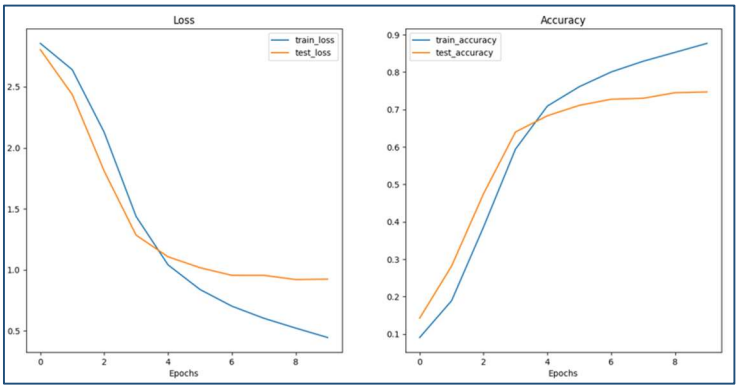


Figure 2 TinyVGG experiment 3

- 4
  Deferent optimizer, Adam was replaced with SWG on which Adam is based.
  Result: Quite unexpected massive drop in performance. Accuracy dropped almost to 0. As Adam was efficient enough, no further attempt to fix or improve functioning with SWG was made.

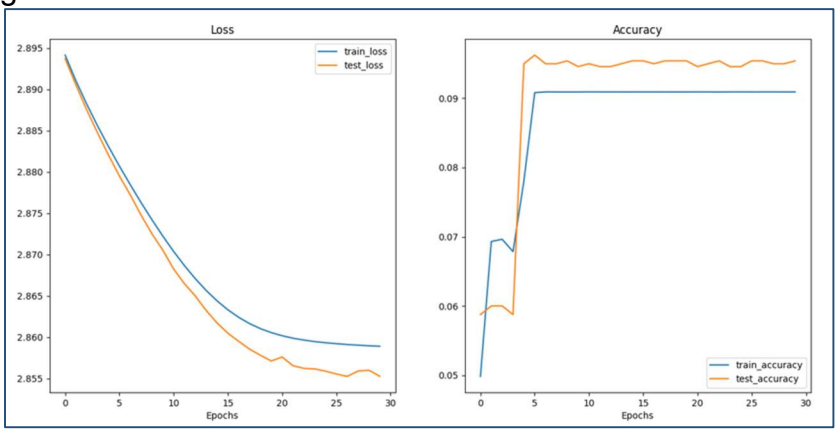


Figure 3 TinyVGG experiment 4

- 5
  Based on previous good results from model 3, training session was made with increased set of data by increasing epochs and batch size.
  Result: Test loss doubled but train loss dropped. No test accuracy was gained but train accuracy hit almost 100%. It means that model was not learning properly and progress was blocked on certain threshold.

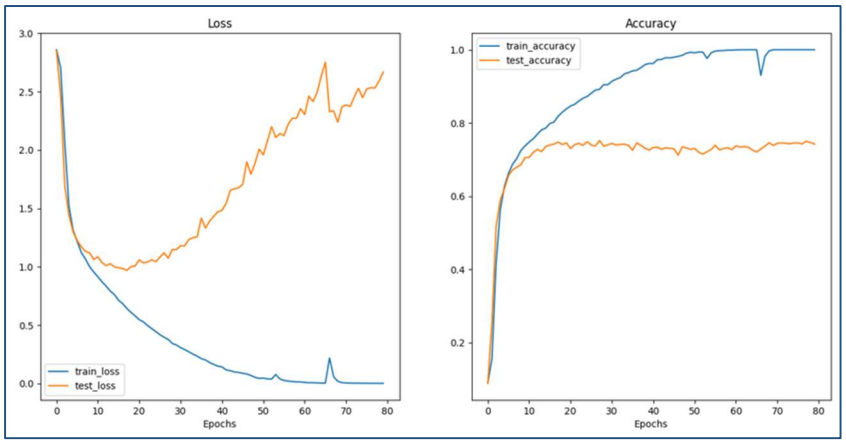


Figure 4 TinyVGG experiment 5

- 6
  Dropout was introduced.
  Result: Mostly negative. All statistic worsened, the reason could be improper usage of dropout.

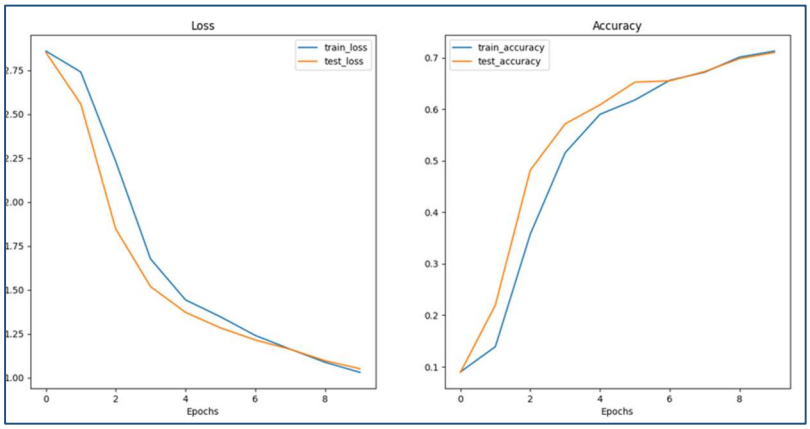


Figure 5 TinyVGG experiment 6

- 7
  Attempt with larger images. Image size was increased 4 times.
  Result: No significant gain from this experiment. However it showed that increase in image size will probably boost efficiency by some value.

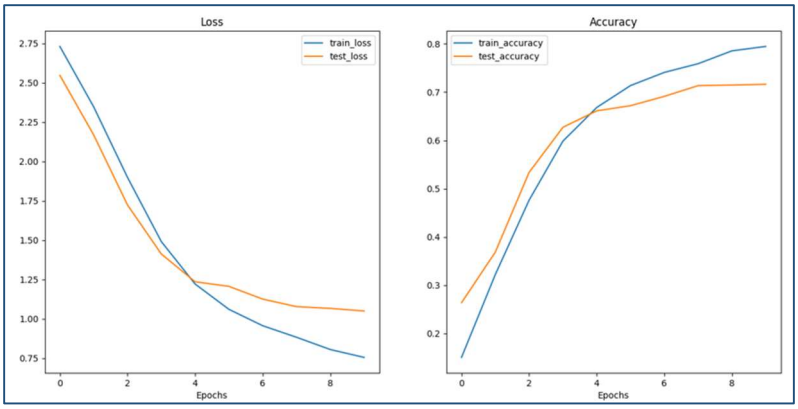


Figure 6 TinyVGG experiment 7

- 8
  New convolution layers were added to the model. Is was assumed that by this action more details will be taken under consideration during the training session.
  Result: Effect were quite similar to increasing image size. Training time increased. Drop in train loss was only real gain. However, this approach might lead to overfitting.

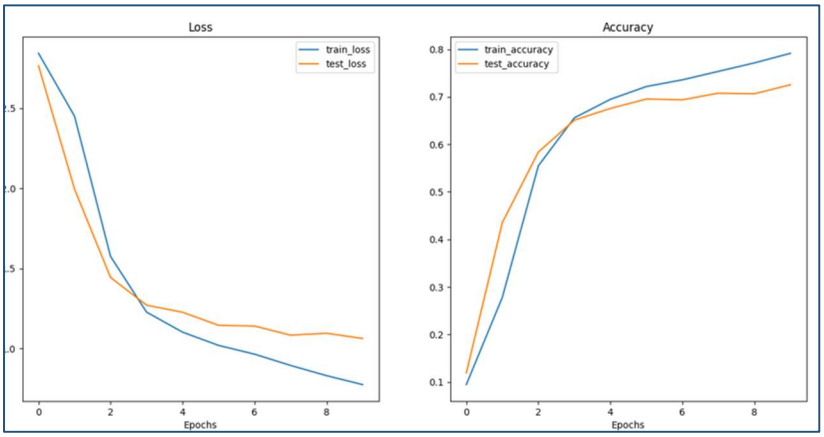


Figure 7 TinyVGG experiment 8

- 9
  Increased dropout percentage of dropped connections and number of dropout layers.
  Result: No improvements.

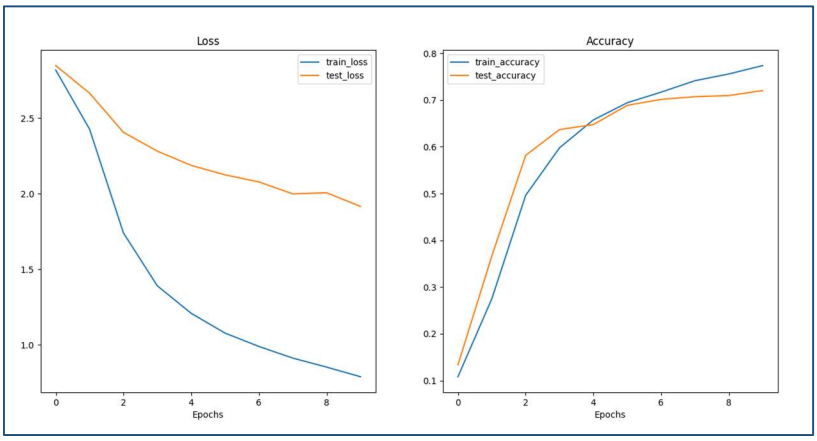


Figure 8 TinyVGG experiment 9

- 10
  Further dropout modifications
  Result: Accuracy and loss worsened. The possible reason being that when introducing dropout, more training is needed as naturally, less neurons are active during certain epoch.

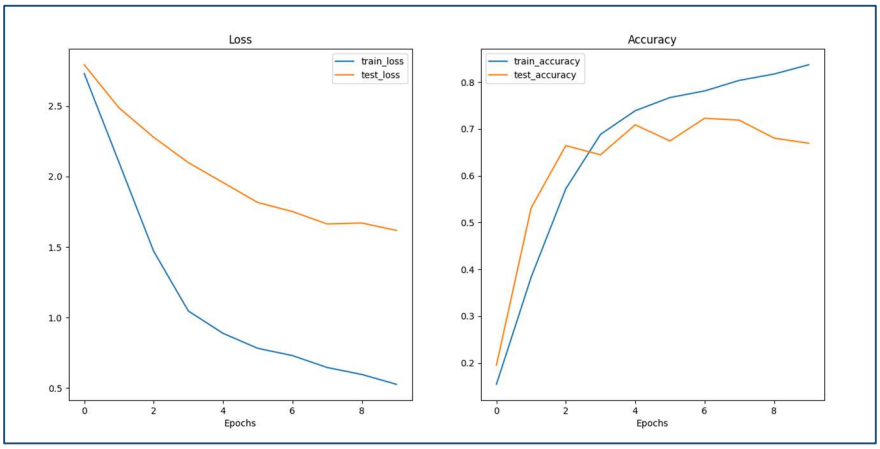


Figure 9 TinyVGG experiment 10

- 11
  Decrease in learning rate and increase in weight loss (optimizer)
  Result: For the first time model achieved test accuracy higher than 75%. The problem of high test loss still remained.

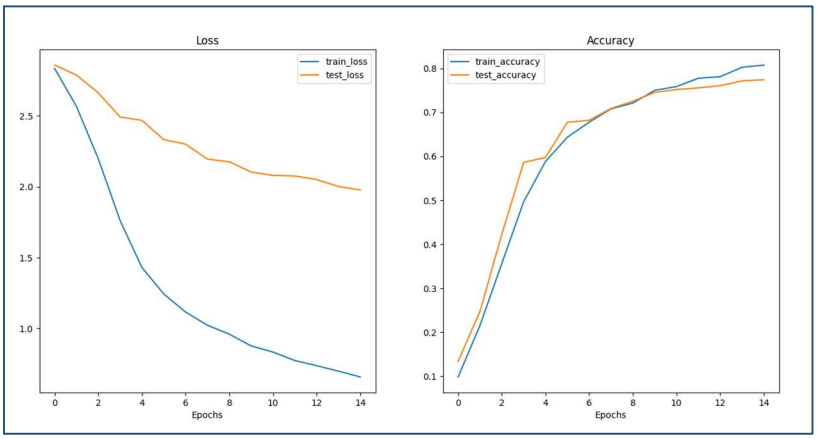


Figure 10 TinyVGG experiment 11

- 12
  Attempt to scale the model down by removing two convolution layers. Simplifying the model might prevent overfitting and increase accuracy.
  Result: Training results were quite satisfactory. Visible drop in test results. It might be not the best direction to develop the model.

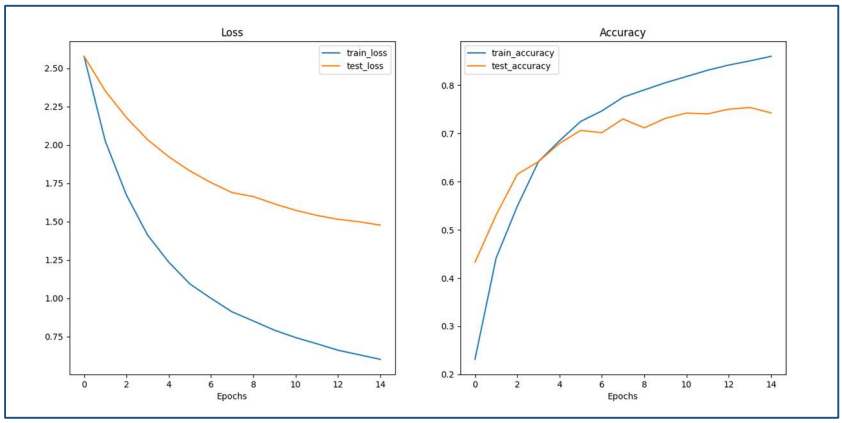


Figure 11 TinyVGG experiment 12

- 13
  Dropout layers increased while its strength was decreased.
  Result: Performance gain, small but it showed right direction and confirmed earlier hypothesis in terms of dropout usage.

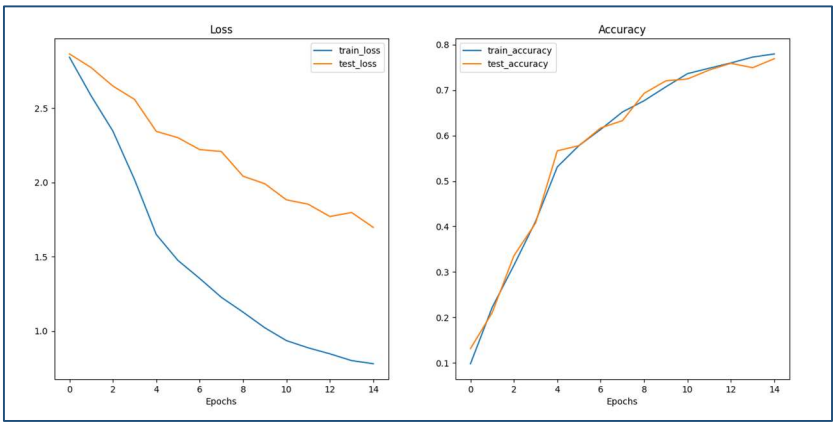


Figure 12 TinyVGG experiment 13

- 14
  Different activation functions. In place of ReLu, SeLu was introduced.
  Result: As train accuracy and loss stayed on similar level, test results were much worse. Attempt assumed as a step back.

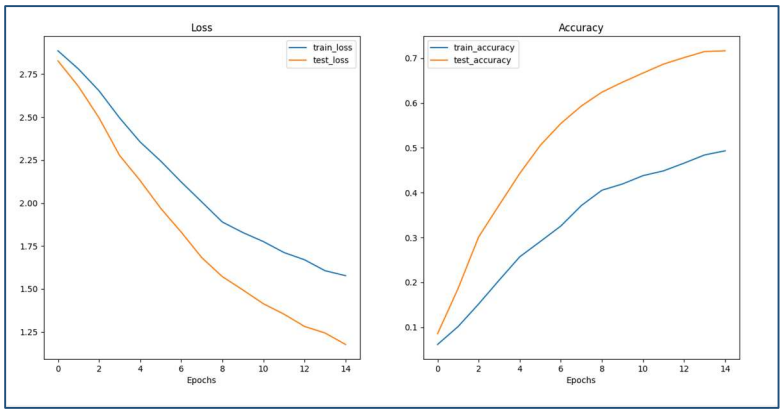


Figure 13 TinyVGG experiment 14

- 15
  Model 13, but trained much longer.
  Result: As expected, accuracy of the model was greatly increased, while test accuracy overcome threshold of 80%, train accuracy hit about 95%. Further training didn't bring any improvements.

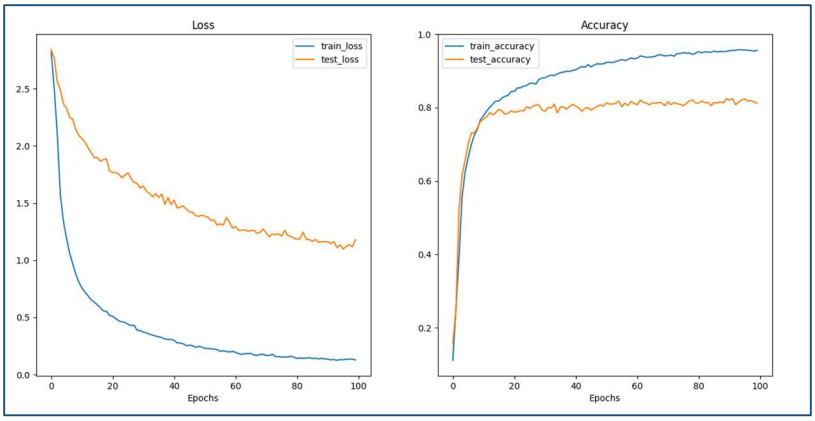


Figure 14 TinyVGG experiment 15

Summary of results:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Train accuracy	86	88	88	0,1	1	72	80	79	80	76	83	90	77	70	95
Test accuracy	72	74	73	0,09	75	72	70	73	74	75	77	73	76	45	80
Train loss	0,55	0,5	0,45	2,86	0,05	1,1	0,73	0,3	0,3	0,3	0,3	0,4	0,2	1,2	0,1
Test loss	1,2	1,15	1	2,858	2,6	1,13	1,12	1,15	2	2,2	2,1	1,55	1,7	1,65	1,4

Table 2 Results of experiments with TinnyVGG

Conclusions:

It was observed that some of images of data set are faulty, they were cropped in such way that important information was lost. Sometimes fingers were out of frame. It might have been a barrier that at some point model couldn't overcome.
Further training and changing of parameters didn't brought any better results or conclusions. 15 model was the best in terms of accuracy. Third one had objectively best values of loss.
While searching for best parameters, it was observed that changes of them mostly lead to small changes, either improvements or deterioration.
Another reason why, accuracy didn't improve further was small complexity of the model. That's why later, the RasNet18 was used.

Experimenting with RasNet18 architecture

As RasNet is much more complex architecture, experience taken from optimizing TinyVGG came in handy. During training of this model, emphasis was put on different things like image preprocessing, optimizer parameters, length of the training and dropouts.

RASNET18	1	2	3	4	5	6	7	8	9
Dropout	0,8	0,8	0,8	0,8	0,8	0,8	0,8	0,9	0,9
Batch	64	64	64	64	64	64	64	64	64
Epochs	15	15	15	15	15	25	25	25	25
Pre-processing	resize	resize	resize, brightness jitter(0.9)	resize, brightness jitter(0.9), contrast	resize, contrast up(0.4)	resize, contrast up(0.6)	resize	resize	resize, contrast up(0.5)
Image size	224x224	224x224	224x224	224x224	224x224	224x224	224x224	224x224	224x224
Learning rate	0,0001	0,0001	0,0001	0,0001	0,00005	0,00005	0,00005	0,00005	0,00005
Optimizer	Adam	Adam	Adam	Adam	Adam	Adam	Adam	Adam	Adam
Weight loss	0,01	0,1	0,01	0,02	-	-	-	-	-

Table 3 Experiments with RasNet

- 1
  Initial version of the model. Learning rate was set to low value of 0.0001 with some weight decay. By using conclusions from the previous model, batch and epochs was increased from the beginning. Image size was a result of a compromise between 256x256 and 128x128.
  Result: First results already were much better than anything form TinyVGG. Only test accuracy hadn't had much of the upgrade. There was another problem, high instability during training occurred and as a result, graph was really inconsistent as values changed drastically with each iteration.

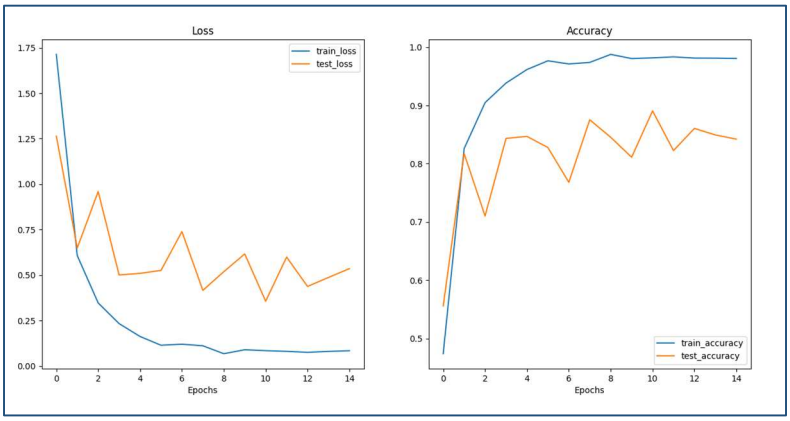


Figure 15 RasNet experiment 1

- 2
  Weight loss was increased.
  Result: Minor improvement in test loss.

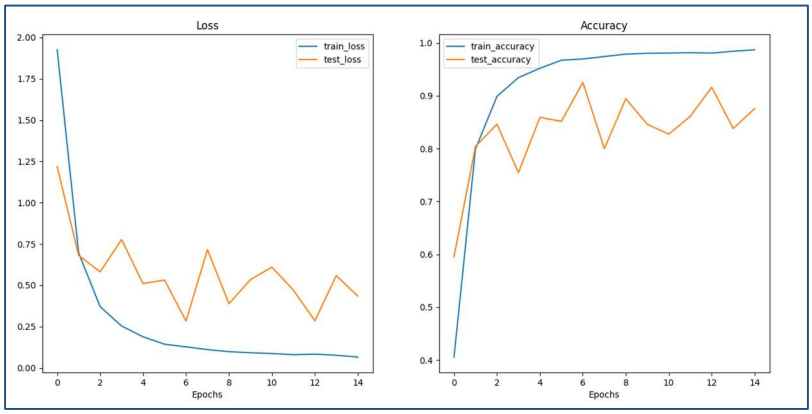


Figure 16 RasNet experiment 2

- 3
  In preprocessing pipeline, brightness jitter was added. This function randomizes brightness for each picture, spread is controlled by passed parameter.
  Result: Jittering brought negative effect on overall performance.

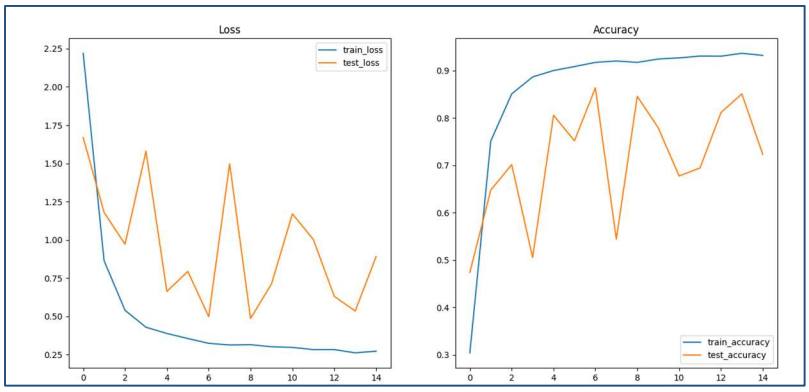


Figure 17 RasNet experiment 3

- 4
  Contract of the images was statically increased by const value. Brightness jitter wasn't changed. Weight loss was decreased to 0.02.
  Result: This try yielded better results than previous one showing that contrast might be good way.

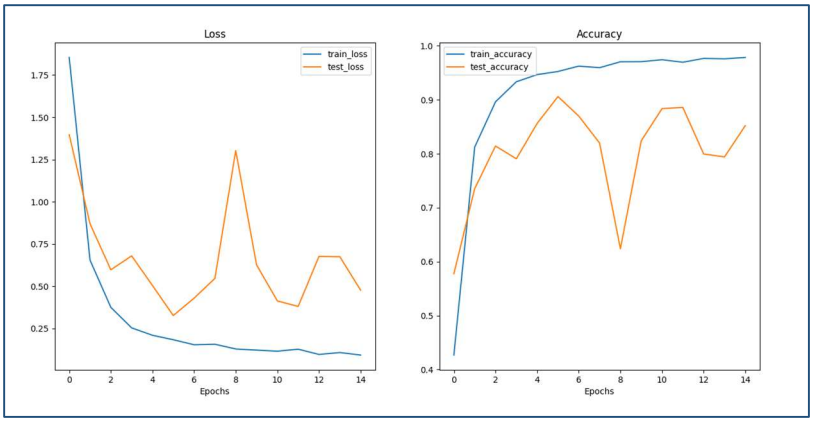


Figure 18 RasNet experiment 4

- 5
  Contrast was decreased slightly. Jittering of brightness and weight decay was removed, while learning rate in optimizer was lowered twice.
  Result: Accuracy was increased by few meaningful percent. Loss also was improved. Many parameters were edited so it was hard to say which of them made the biggest difference. Pots have softened, possibly both weight loss and additional preprocessing were the cause of this behaviour.

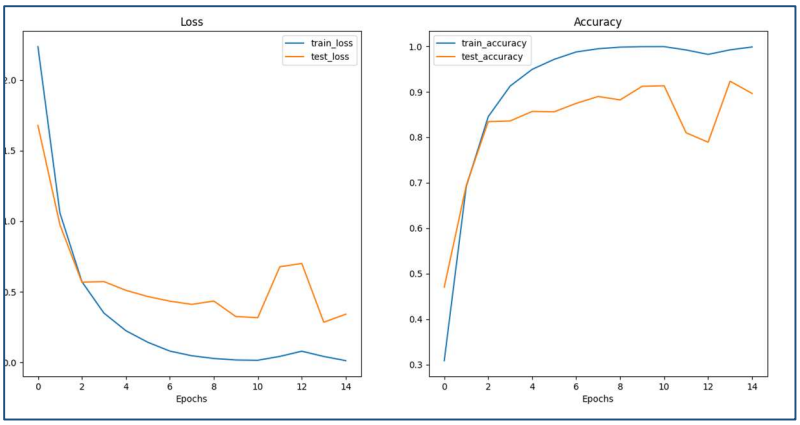


Figure 19 RasNet experiment 5

- 6
  Contrast returned to initial value. Number of epochs was increased to better analyse the results.
  Result: No drastic changes. Only plots started to get uneven again.

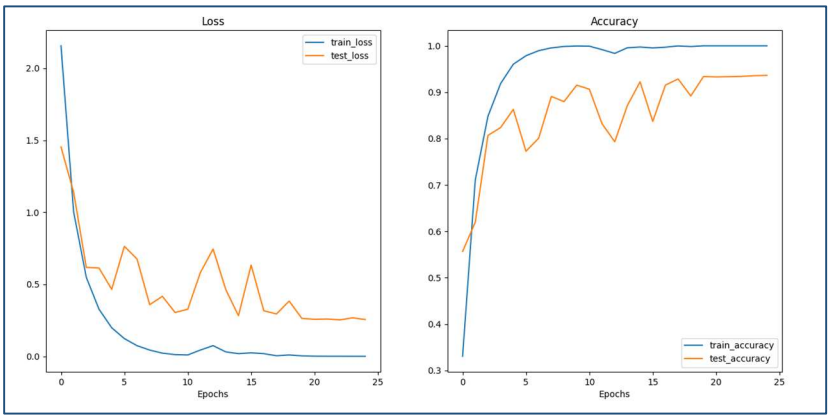


Figure 20 RasNet experiment 6

- 7
  In this try, all preprocessing functions have been dropped (excluding resize).
  Result: Really good results, close to perfection. Increase in accuracy and decrease in loss. This try suggests that no preprocessing method is needed.

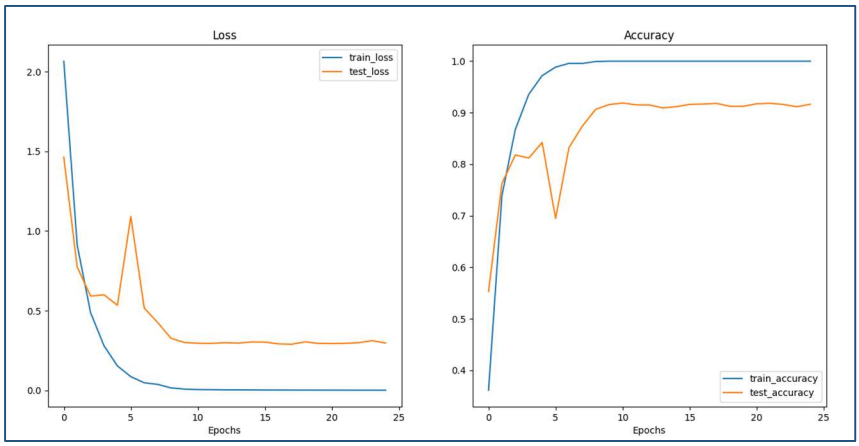


Figure 21 RasNet experiment 7

- 8
  Dropout was increased to 0.9.
  Result: This session brought best results. Nearly 100% accuracy and minimal loss. Plots were relatively smooth, which was an initial problem of rasnet.

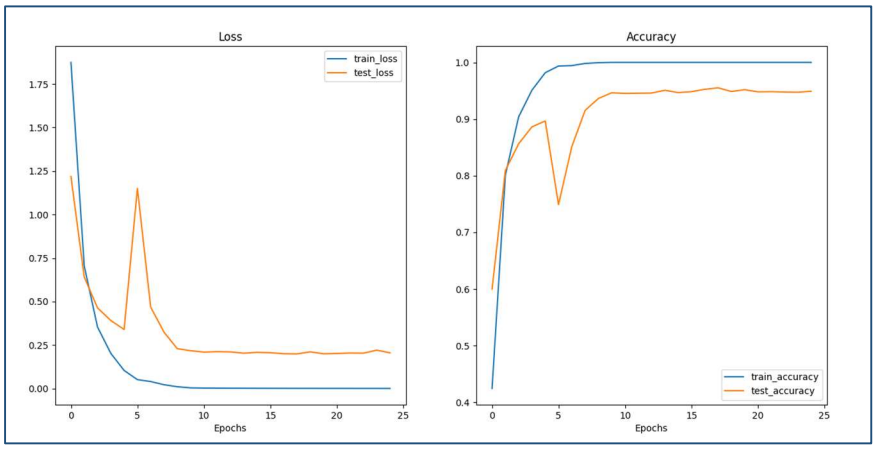


Figure 22 RasNet experiment 8

- 9
  Contrast was added again to see if it will help in any way.
  Result: There was little if no difference in resulting measurements. However, plots became unstable again, which confirms that preprocessing was one of things that generated this issue.

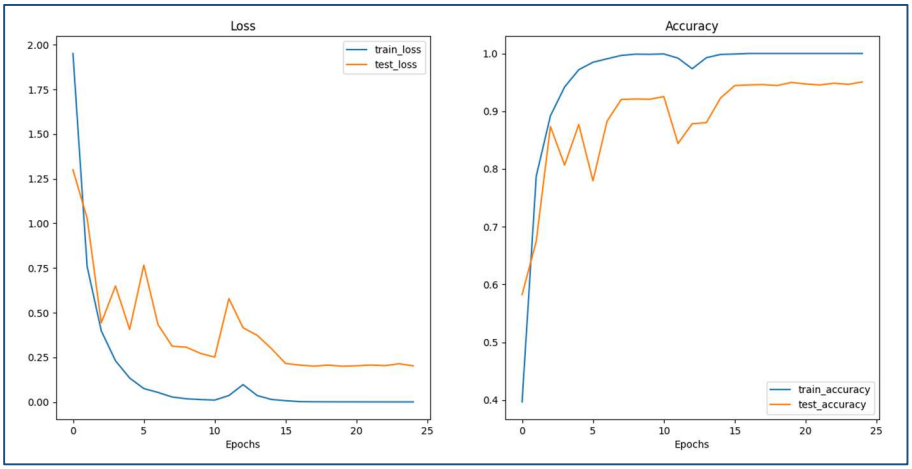
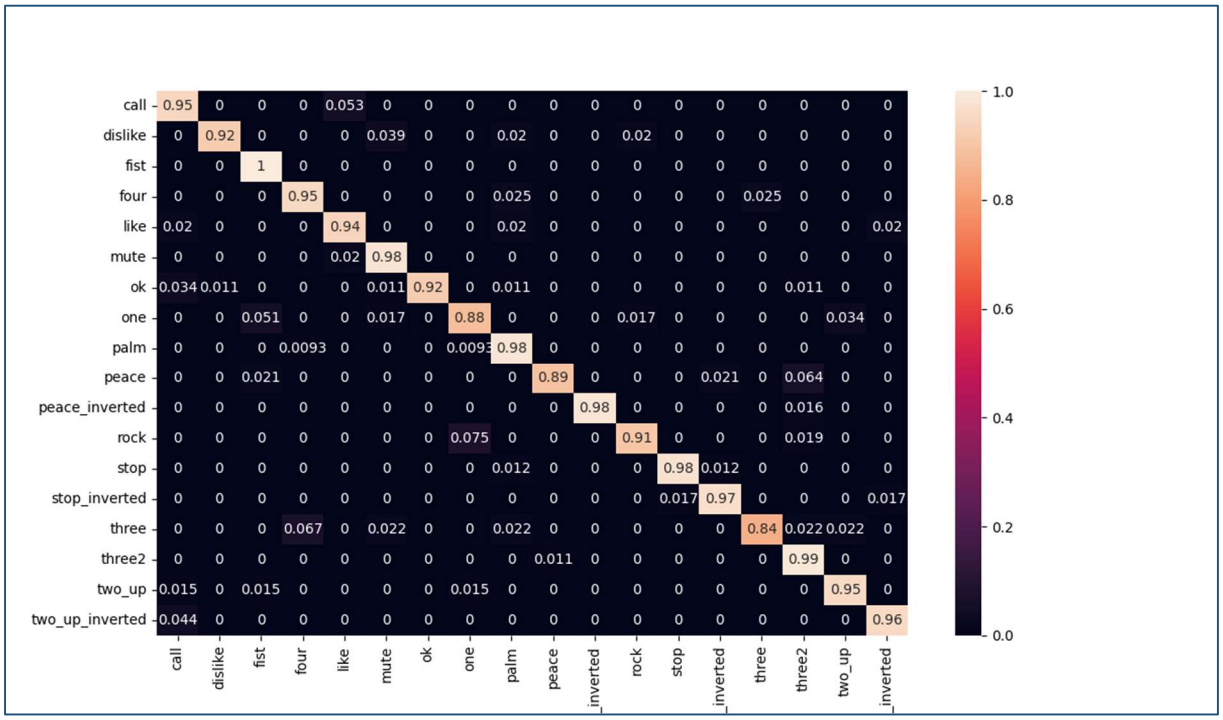
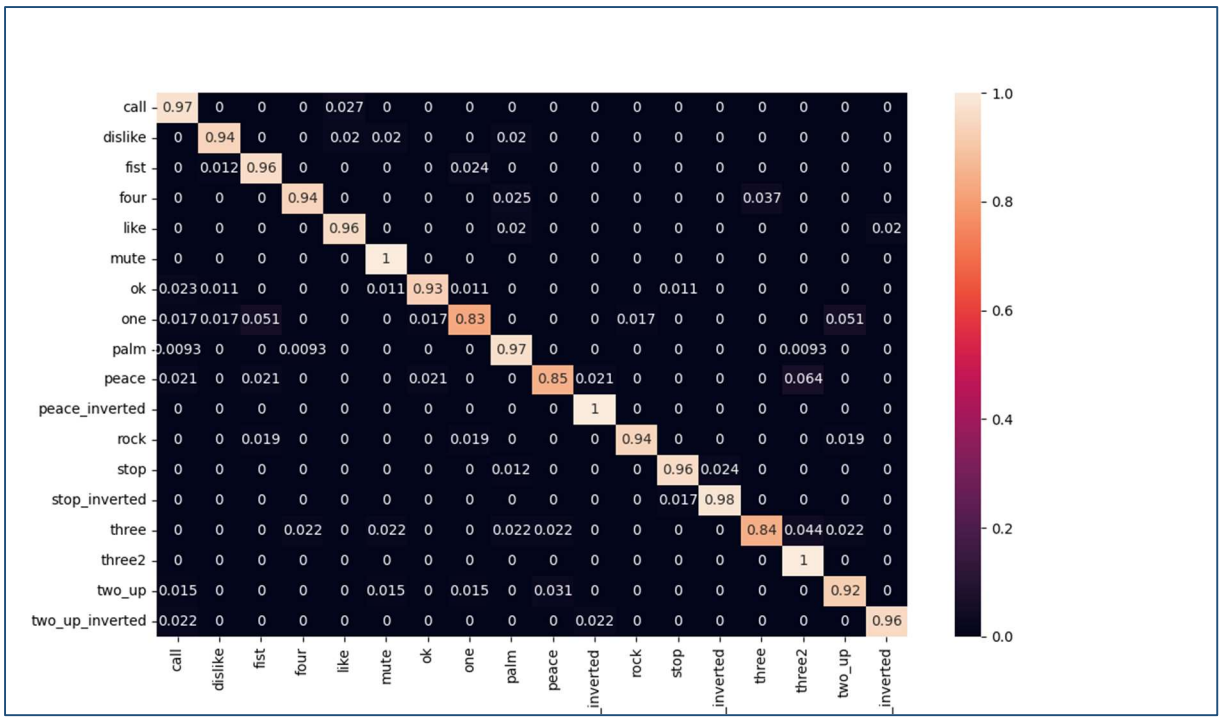


Figure 23 RasNet experiment 9

Confusion matrixes:



Picture 9 Confusion matrix for RasNet experiment 8



Picture 10 Confusion matrix for RasNet experiment 9

Comment:

There are more similarities between these two matrixes than differences.
Most notable observations are that gesture “three” was often mistaken as “four” (Intuitively it should be mistaken for “three2” more often).
“One” and “peace” have relatively low accuracy as they have been observed most often as faulty data where fingers were out of frame.

Additional note:

Modifying images to grayscale had little if none positive effect on the results, sessions from this modification are not present here.

Summary of results:

	1	2	3	4	5	6	7	8	9
Train accuracy	98	98	93	97	99	99	100	100	100
Test accuracy	84	86	72	84	92	93	92	94	93
Train loss	0,08	0,08	0,27	0,12	0,1	0,05	0,02	~0	~0
Test loss	0,84	0,5	0,81	0,55	0,43	0,3	0,3	0,23	0,25

Table 4 Results of experiments with RasNet

Conclusion:

There is a weird peak on both plots around 6-7 epoch on every model (clearly visible on graph 8). It might be the place where most of the faulty data came through the model (even that after split data should be shuffled anyway).
After numerous experiments, data shows that no preprocessing was needed, objectively increasing contrast, brightness or changing rgb scale to grayscale given worse results.
Thanks to experience taken from previous architecture, modifying this architecture was much less random and as it can be observed on summary, it resulted in nearly continuous improvement with every modification.
Test accuracy might have been possibly even higher if more data augmentation was introduced and the model was subject to more training.
After analysing all graphs and visual testing, it is clear that imperfections in dataset actively influence quality of the model.

Final conclusions:

This project showed that there are two important factors that must be considered: quality of the chosen dataset as it will greatly impact whole workflow and results later, and properly configured model, it is a good practice to start building own models by basing on working architectures.
Another very important thing is consistency during both training and testing. Lack of it will most certainly lead to wrong assumptions and conclusions creating a snow ball effect and useless model.
In terms of challenges met along development: PyTorch has decent documentation, but some of its aspect aren't clearly visible or explained at first. Especially saving and loading models were problematic. Models are also prone to corruption, because slight changes in code or file header might make irreversible damage.

Sources:

https://poloclub.github.io/cnn-explainer/
https://pytorch.org/docs/stable/index.html
https://www.tutorialspoint.com/pytorch/index.htm
https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial2/Introduction_to_PyTorch.html
https://realpython.com/python-ai-neural-network/

Link to project:

https://github.com/Eques72/HandGestureClassificaton