



Politechnika Śląska

**Katedra Grafiki, Wizji komputerowej
i Systemów Cyfrowych**



Academic year			Group	Section
2022/2023	SSI	BIAI	ISMIP	1
Supervisor:	dr inż. Grzegorz Baron		Classes: (day, hour)	
Section members: emails: macicab865@student.polsl.pl	Maciej Caban		Monday	
			13:15 – 16:15	

Project card

Subject:

American Sign Language Recognition

Table of contents:

1. Project topic	3
2. Analysis of the task	3
2.1 Possible Approaches:	3
2.2 Datasets:	4
2.2.1 Sign Language MNIST	4
2.2.2 American Sign Language	5
2.2.3 27 Class Sign Language Dataset	5
2.2.4 ASL Alphabet	6
2.2.5 Sign Language Digits Dataset	7
2.2.6 Sign Language for Numbers	7
2.2.7 Sign Language for Alphabets	8
2.2.8 Dataset Selection:	9
2.3 Analysis of Available Tools/Frameworks/Libraries:	9
2.3.1 Tools selected:	10
2.4 IDE:	11
2.4.1 IDE Selected:	12
3. Internal and external specification of the software solution	12
3.1 Scripts:	12
3.1.1 Load data	12
3.1.2 Scale data	12
3.1.3 Split data	13
3.1.4 Create training model	13
3.1.5 Compile model	15
3.1.6 Train model	15
3.1.7 Plot performance	15
3.1.8 Evaluate model	15
3.2 Data structures	15
3.3 UI	16
4. Experiments	16
4.1 Training few models on main dataset	16
4.1.1 Experiment description	16
4.1.2 The prepared models are presented as follows:	16
4.1.3 The training outcomes are presented below:	17
4.1.4 Training effects:	24
4.1.5 Testing models:	24
4.1.6 Conclusions from the experiment:	24
4.2 Experiment with MNIST dataset	25
4.2.2 Training effects	25
4.2.3 Testing effects:	27
4.2.4 Results:	27
5. Conclusions	27
6. Future work	27
7. Sources	28
8. Links to project:	28

1. Project topic

The topic of the project was to prepare a neural network capable of recognizing letters and numbers from American Sign Language. The recognition of signs will be done from images. The letters J and Z will be excluded from the recognition due to them being shown through movement.

2. Analysis of the task

2.1 Possible Approaches:

Image Classification:

One approach is to treat the recognition problem as an image classification task. This involves training a deep learning model, such as a convolutional neural network (CNN), on a large dataset of ASL gesture images. The model can learn to identify specific hand shapes, movements, and patterns associated with different ASL signs.

Pros:

- CNNs are highly effective in learning and recognizing patterns in images.
- This approach can handle variations in lighting conditions and background noise to some extent.
- It can be trained on large datasets, enabling better generalization.

Cons:

- Static images may not capture the dynamic nature of sign language, which includes movements and transitions between signs.
- Recognition accuracy may be affected by occlusions, variations in hand orientations, and individual differences in signing styles.

Pose Estimation:

Another approach is to use pose estimation techniques to capture the skeletal structure of the hand and fingers. This involves extracting key points or landmarks from video frames using methods like OpenPose or MediaPipe. The position and movement of these points can then be used to recognize ASL gestures.

Pros:

- Pose estimation can capture the dynamic nature of sign language, including hand movements and transitions between signs.
- It can handle variations in hand orientation and some occlusions.
- The use of landmarks reduces the complexity of the recognition task compared to processing raw images.

Cons:

- It may be challenging to accurately estimate hand poses in real-time due to occlusions, lighting conditions, and complex hand configurations.
- Variations in individual signing styles can impact recognition accuracy.

Conclusion:

The project does not involve gesture recognition that requires motion. Therefore, a better solution would be Image Classification, which is simpler and perfectly suited to this topic.

2.2 Datasets:

The datasets were downloaded from the Kaggle platform, which provides a wide selection of appropriately balanced and numerous datasets. The following datasets were subjected to analysis:

2.2.1 Sign Language MNIST (<https://www.kaggle.com/datasets/datamunge/sign-language-mnist>)

Dataset contains images of letters A-Z, excluding J and Z. The images consist of 784 pixels (28x28). The dataset includes a total of 27,455 images in the training set (slightly over 1,100 images per letter) and 7,172 images in the test set (approximately 300 per letter). The dataset is saved in CSV format, where each row represents a consecutive image. The first column contains information about the letter represented by the image, and the subsequent columns contain the values of the corresponding pixels (ranging from 0 to 255).



Image 1: show example of images from MNIST dataset


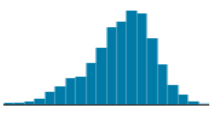
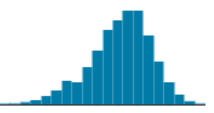
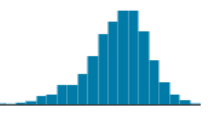
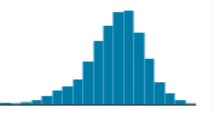
# label	# pixel1	# pixel2	# pixel3	# pixel4
				
0	0	0	0	0
24	255	255	255	255
3	107	118	127	134
6	155	157	156	156
2	187	188	188	187
2	211	211	212	212
13	164	167	170	172
16	161	168	172	173
8	134	134	135	135
22	114	42	74	99
3	169	174	176	180

Diagram 1: show fragment of csv file

The dataset was rejected because it did not meet the requirements of the project topic as it only contained letters. However, it was still chosen as an additional dataset for conducting supplementary experiments (see 4.2).

2.2.2 American Sign Language (<https://www.kaggle.com/datasets/kapillondhe/american-sign-language>)

The dataset contains approximately 168,000 images (5 GB) divided into 28 classes, with 6,000 images per class. In addition to letters, it also includes symbols representing 'space' and 'nothing' (images of an empty hand or a plain wall). The images are in color and depict a hand showing a letter against a single-colored background.

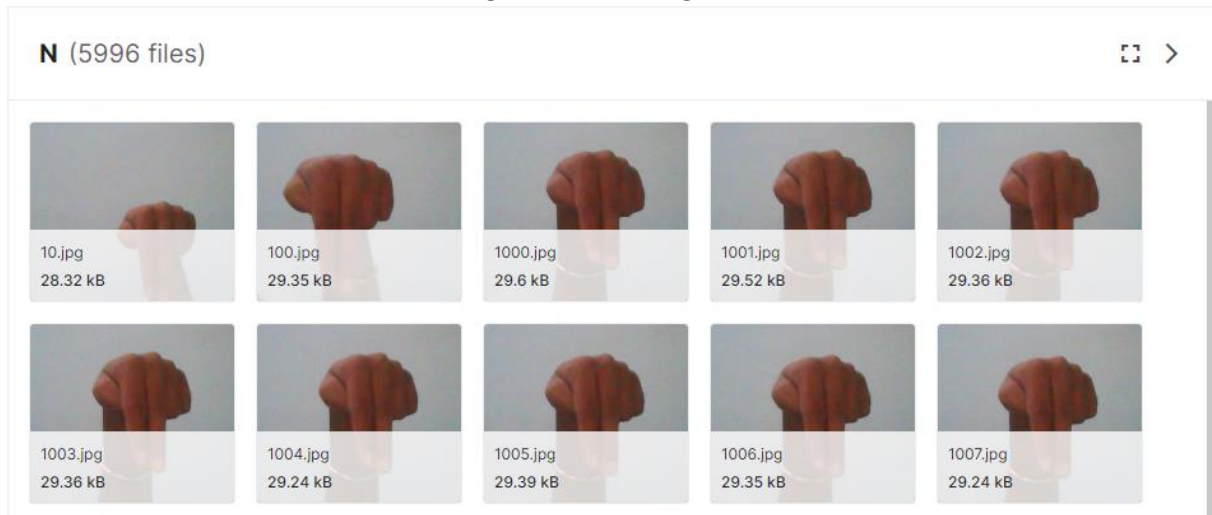


Image 2: show example images from dataset

The dataset was rejected because it did not meet the requirements of the project topic as it only contained letters.

2.2.3 27 Class Sign Language Dataset (<https://www.kaggle.com/datasets/ardamavi/27-class-sign-language-dataset>)

The dataset was created by processing American Sign Language-based photographs collected from 173 volunteer individuals. The images are saved in two files, X.npy and Y.npy, with a combined size of less than 4.5 GB. The dataset represents 27 American Sign Language (ASL) characters or gestures.



Image 3: show example images from dataset

The dataset was rejected because it did not meet the project's requirements as it did not contain digits. However, this dataset can still be utilized for testing a pre-trained network. The diversity of individuals involved in preparing the images for this dataset would serve as a good test for the trained model. It would allow assessing how well the model performs when letters are being performed against various backgrounds, by different people, under different conditions.

2.2.4 ASL Alphabet (<https://www.kaggle.com/datasets/grassknoted/asl-alphabet>)

The data set is a collection of images of alphabets from the American Sign Language, separated in 29 folders which represent the various classes. The training data set contains 87,000 images which are 200x200 pixels. There are 29 classes, of which 26 are for the letters A-Z and 3 classes for SPACE, DELETE and NOTHING.

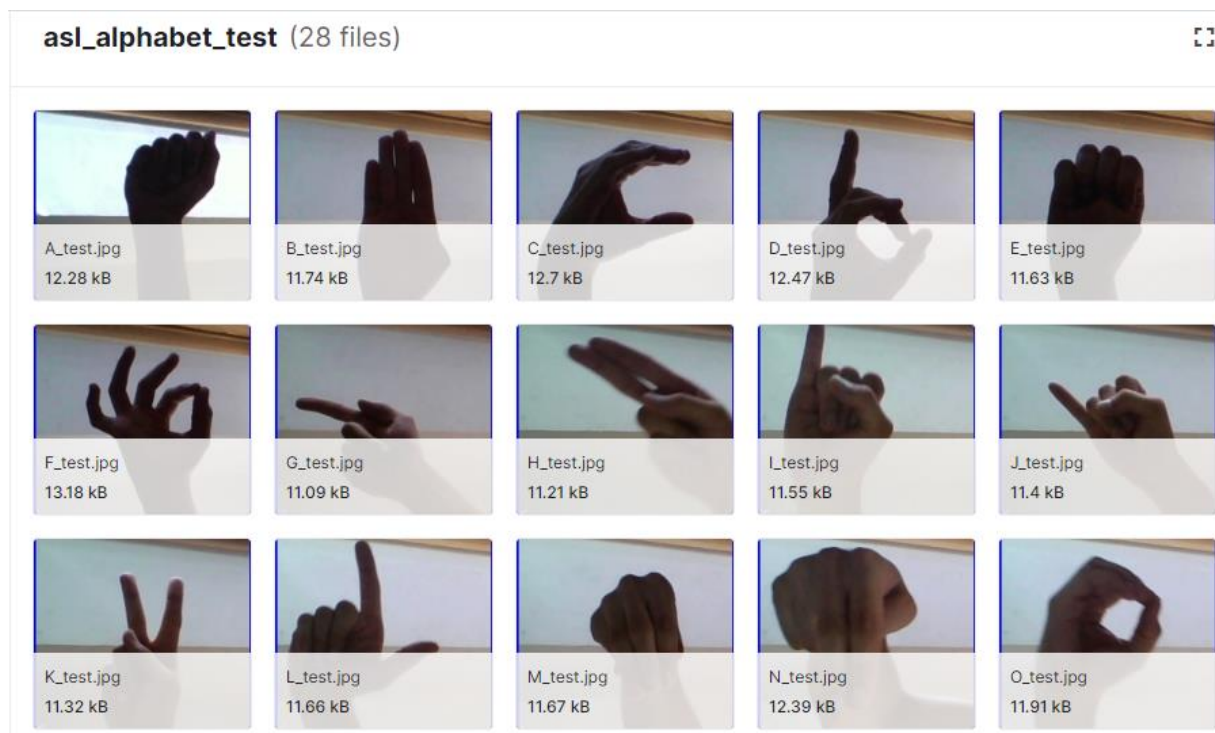


Image 4: show example images from dataset

The dataset was rejected because it did not meet the project's requirements as it did not contain digits.

2.2.5 Sign Language Digits Dataset (<https://www.kaggle.com/datasets/ardamavi/sign-language-digits-dataset>)

The dataset contains images representing digits 0-9 performed in American Sign Language (ASL).

Image size: 64x64. Color space: Grayscale. File format: npy. Number of participant students: 218. Number of samples per student: 10



Image 5: show example images from dataset

The dataset was rejected because it did not meet the project's requirements as it did not contain digits.

2.2.6 Sign Language for Numbers (<https://www.kaggle.com/datasets/muhammadkhalid/sign-language-for-numbers/code>)

The dataset contains images of 11 classes: digits 0-9 and 'nothing'. The total number of images is 16,500, with 1,500 images per class. The images are in color and taken against a single-colored background.



Image 6: show example images from dataset

Despite the absence of letters, the dataset was chosen (refer to 2.3.7).

2.2.7 Sign Language for Alphabets (<https://www.kaggle.com/datasets/muhammadkhalid/sign-language-for-alphabets>)

The dataset contains images of 27 classes: letters a-z and 'nothing'. The total number of images is 40,500, with 1,500 images per class. The images are in color and taken against a single-colored background.



Image 7 show example images from dataset

2.2.8 Dataset Selection:

The datasets described in points 2.3.6 and 2.3.7 were chosen for training the neural network. Combined, they meet the project's requirements. Both datasets were created by the same author, so the network will not have any issues with letters being performed by one person and digits by another. This will help avoid the network learning based on hand appearance (for example skin color).

2.3 Analysis of Available Tools/Frameworks/Libraries:

TensorFlow:

TensorFlow is a popular deep learning framework widely used for training and deploying CNN models. It offers a comprehensive set of tools and APIs for building and training neural networks, including CNN architectures. TensorFlow provides GPU acceleration for faster training and supports model deployment on various platforms.

PyTorch:

PyTorch is another widely adopted deep learning framework known for its dynamic computational graph and user-friendly interface. It provides extensive support for CNNs and allows efficient model development and training. PyTorch also offers GPU acceleration and seamless integration with Python, making it a suitable choice for ASL recognition.

Google Colab:

Google Colab, short for Google Colaboratory, is a cloud-based platform provided by Google that offers a free environment for running and executing Python code. It allows users to write and execute Python code directly in a web browser without requiring any setup or installation. Google Colab provides free GPU/TPU support and Jupyter Notebook-like interface where users can create, edit, and run code cells.

Keras:

Keras is a high-level deep learning library that runs on top of TensorFlow or other backend frameworks. It provides a user-friendly interface for designing and training CNN models. Keras simplifies the process of building and experimenting with different architectures, making it suitable for rapid prototyping in ASL recognition projects.

OpenCV:

OpenCV (Open Source Computer Vision Library) is a powerful library widely used for computer vision tasks. It offers various image processing and computer vision algorithms that can be utilized in

preprocessing steps, such as image enhancement, noise reduction, and segmentation. OpenCV provides an extensive set of functions for handling image data, making it valuable in ASL recognition projects.

Pretrained models (VGG, ResNet, Inception etc):

Pretrained models are models that have been trained on large datasets (which would be impossible without specialized datasets and hardware), allowing them to be used as a starting point for transfer learning. By adapting these models to a specific domain, they can be utilized effectively.

NumPy

NumPy (Numerical Python) is a fundamental library in Python for scientific computing and data manipulation. It provides support for handling large, multi-dimensional arrays and matrices, along with a wide range of mathematical functions to perform operations on these arrays efficiently.

Pyplot

Pyplot is a module within the Matplotlib library, which is a popular plotting library in Python. It provides a high-level interface for creating a wide range of static, animated, and interactive visualizations.

2.3.1 Tools selected:

For this project, we will select the following tools and approach:

TensorFlow: We will utilize TensorFlow as the deep learning framework for training and deploying CNN models. TensorFlow offers excellent GPU acceleration and provides a rich ecosystem of tools and resources for building and optimizing CNN architectures.

Keras: To simplify model development and experimentation, we will leverage Keras, which is a user-friendly high-level API running on top of TensorFlow. Keras will enable us to rapidly prototype and iterate CNN architectures, making it efficient for ASL recognition tasks.

OpenCV: We will incorporate OpenCV for image preprocessing and manipulation. OpenCV's functionalities will allow us to perform various image processing operations, such as enhancing image quality, reducing noise, and segmenting hand regions, which are crucial for accurate ASL recognition.

NumPy: it will provides support in operating on large arrays of data

PyPlot: we will use PyPlot to improve visualization of results CNN work, like creating a diagrams.

2.4 IDE:

JupyterLab:

- Notebook-based Environment: JupyterLab is built around the concept of notebooks, which are interactive documents combining live code, visualizations, and explanatory text. It allows for the creation, execution, and sharing of notebooks containing code, Markdown, and visualizations.
- Interactive Data Exploration: JupyterLab excels at interactive data exploration and analysis. It provides an interactive and iterative workflow, allowing users to run code cells individually and visualize the results immediately. This is particularly useful for prototyping, experimenting, and presenting data-driven insights.
- Rich Display Capabilities: JupyterLab supports the display of various media types, including images, plots, tables, HTML, and more. It makes it easy to create and visualize rich content within notebooks, enabling effective communication of data analysis.
- Integration with Scientific Libraries: JupyterLab seamlessly integrates with popular scientific libraries such as NumPy, Pandas, Matplotlib, and Scikit-learn. This integration allows for efficient data manipulation, analysis, and visualization within the notebook environment.

Visual Studio Code:

- General-purpose IDE: Visual Studio Code is a versatile and extensible integrated development environment (IDE) that supports a wide range of programming languages and workflows. It provides a feature-rich environment for coding, debugging, and version control.
- Code Editing and Productivity: VS Code offers powerful code editing capabilities, including code autocompletion, linting, formatting, and intelligent code suggestions. It has a large extension ecosystem that allows users to customize and extend the functionality of the IDE according to their specific needs.
- Integrated Debugging: VS Code provides built-in debugging support for various programming languages. It allows users to set breakpoints, inspect variables, and step through code to identify and fix issues in their programs.
- Version Control Integration: VS Code has seamless integration with popular version control systems like Git. It provides a visual interface for managing repositories, committing changes, and resolving merge conflicts.
- Extensibility: VS Code's extensibility is one of its standout features. It supports a vast array of extensions that enhance the functionality and support for different languages, frameworks, and tools. Users can customize their workflow and add features specific to their projects.

2.4.1 IDE Selected:

The choice was made for JupyterLab due to its clarity and ease of integrating with the necessary tools. Additionally, on the used computer, Visual Studio Code had issues finding the Python path in the environment variables. Attempting to fix this error took a lot of time and did not yield the expected results.

3. Internal and external specification of the software solution

3.1 Scripts:

3.1.1 Load data - The "image_data_set_from_directory" function provided in the "keras.utils" library was used to load the dataset. It allows for loading data from sorted image folders based on categories, creating a dataset without the need to individually label each image. The images were loaded in a size of 32x32 to save time and accommodate the hardware on which the network will be trained (more details in the experiments section). Then, using the NumPy library, several random images were displayed.

```
#Load data
import numpy as np
from matplotlib import pyplot as plt
data = tf.keras.utils.image_dataset_from_directory('Data1', batch_size=32, image_size = (32,32))
data_iterator = data.as_numpy_iterator()
batch = data_iterator.next()
fig, ax = plt.subplots(ncols=4, figsize=(20,20))
for idx, img in enumerate(batch[0][:4]):
    ax[idx].imshow(img.astype(int))
    ax[idx].title.set_text(batch[1][idx])
```

Found 48000 files belonging to 33 classes.

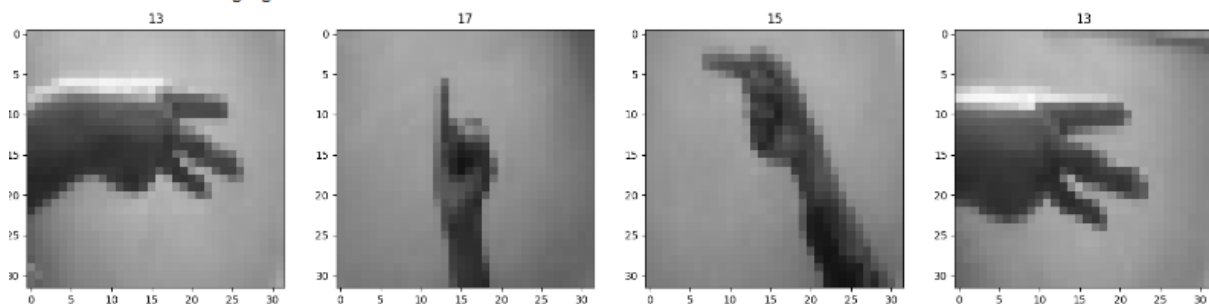


Image 8: show loading data script

3.1.2 Scale data - the next script is responsible for scaling the images. The value of each pixel was divided by 255, which changed its value range from 0-255 to 0-1. Performing this operation allows for saving time during operations on the image, such as training a neural network. Then, using the NumPy library, several random images were displayed to check if the images have been modified, you can compare the images before and after the scaling process. Scaling the pixel values did not visibly alter the images and had no impact on training the neural network. The only effect was the saving of time during the operations performed.

```
# Scale Data
data = data.map(lambda x,y: (x/255, y))
data.as_numpy_iterator().next()
```

```
data_iterator = data.as_numpy_iterator()
batch = data_iterator.next()
fig, ax = plt.subplots(ncols=4, figsize=(20,20))
for idx, img in enumerate(batch[0][:4]):
    ax[idx].imshow(img)
    ax[idx].title.set_text(batch[1][idx])
```

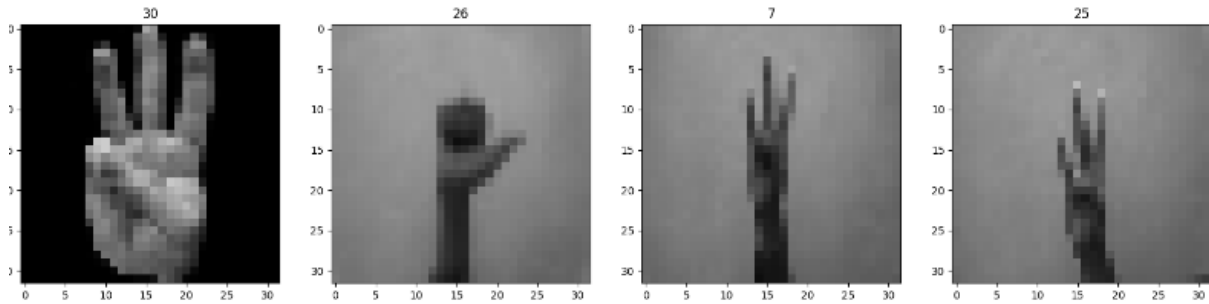


Image 9: show scaling data script

3.1.3 Split data - Next, the dataset was divided into three parts: the training set, the validation set, and the test set. By splitting the data into these three sets, we can ensure that the model is trained, fine-tuned, and evaluated on different datasets. This helps us estimate the model's performance accurately, avoid overfitting to the training data, and obtain reliable metrics for its generalization capabilities.

```
# Split Data
train_size = int(len(data)*.7)
val_size = int(len(data)*.2)
test_size = int(len(data)*.1)

train = data.take(train_size)
val = data.skip(train_size).take(val_size)
test = data.skip(train_size+val_size).take(test_size)
```

Image 10: show splitting data script

3.1.4 Create training model - this script is responsible for configuring the model that will be compiled and trained.

Model was created using the following layers:

- "Conv2D" (2D Convolutional Layer) is a fundamental building block in CNN architectures used for image recognition, computer vision, and other related tasks. Conv2D applies a convolution operation to the input data, which involves sliding a small window (called a kernel or filter) over the input image or feature map. The kernel consists of learnable weights that are adjusted during the training process. The purpose of the convolution operation is to extract local patterns and features from the input data.
- Filters: Conv2D uses a set of filters to convolve with the input data. Each filter captures different features or patterns from the input. These filters act as feature detectors and can learn to detect edges, textures, shapes, or more complex patterns.

- **Padding:** Padding is an optional parameter in the Conv2D layer. It determines whether to add extra pixels around the input image or feature map before performing the convolution operation. Padding helps preserve spatial dimensions and prevent the reduction of spatial information as the filter slides over the input. Common padding options include "valid" (no padding) and "same" (padding to maintain the same spatial dimensions).
- **Activation Function:** Conv2D layers are often followed by an activation function, such as ReLU (Rectified Linear Unit), which introduces non-linearity into the model. The activation function helps the network learn complex representations by introducing non-linear transformations to the convolutional outputs.
- **MaxPooling2D:** operation commonly used in convolutional neural networks (CNNs) to downsample the spatial dimensions of feature maps. It helps reduce the computational complexity and extract important features while maintaining their spatial hierarchy.
- **Batch normalization:** is a technique used in convolutional neural networks to improve the training and performance of the model. It normalizes the intermediate outputs of a layer by adjusting and scaling them, typically applied after the convolution and activation layers. It contributes to faster convergence, better performance, and increased stability of CNN models during training and inference.
- **Dropout:** is a layer used in convolutional neural networks to prevent overfitting and improve the generalization ability of the model. It randomly "drops out" (sets to zero) a certain percentage of the input units or neurons during training.
- **Batch normalization – layer** used in convolutional neural networks (CNNs) to improve the training and performance of the model. It normalizes the intermediate outputs of a layer by adjusting and scaling them, typically applied after the convolution and activation layers.
- **Flatten:** operation is used to reshape the multi-dimensional output from convolutional and pooling layers into a one-dimensional vector. It is typically applied before the fully connected layers in the network.
- **Dense:** these layers play a crucial role in neural networks by enabling the model to learn complex mappings between input and output data. They provide flexibility and expressive power to capture intricate patterns and make accurate predictions.

```
#create training model
model = Sequential()

model.add(Conv2D(64, (3, 3), padding='same', input_shape=(32, 32, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.BatchNormalization())

model.add(Conv2D(128, (3, 3), padding='same', input_shape=(32, 32, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.BatchNormalization())
#model.add(Dropout(0.5))
model.add(Dropout(0.2))

model.add(Conv2D(256, (3, 3), padding='same', input_shape=(32, 32, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.BatchNormalization())

model.add(Flatten())
#model.add(Dropout(0.5))
model.add(Dropout(0.2))
model.add(Dense(1024, activation='relu'))
model.add(Dense(33, activation='softmax'))
```

Image 11: show creating model script

3.1.5 Compile model – In the next step the model was compiled using the Adam optimizer.

```
model.compile(optimizer = 'adam', loss = 'sparse_categorical_crossentropy', metrics = ['accuracy'])
```

3.1.6 Train model - the script begins the training of the model. The training lasted for 20 epochs to fully observe the training effects. It was determined that additional epochs were not necessary (more details in the "Experiments" section).

```
#train
logdir='logs'
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=logdir)
hist = model.fit(train, epochs=20, validation_data=val, callbacks=[tensorboard_callback])
```

Image 12: show training script

3.1.7 Plot performance – they use matplotlib library to visualize effectiveness of training the model.

```
#Plot Performance
fig = plt.figure()
plt.plot(hist.history['accuracy'], color='teal', label='accuracy')
plt.plot(hist.history['val_accuracy'], color='orange', label='val_accuracy')
fig.suptitle('Accuracy', fontsize=20)
plt.legend(loc="upper left")
plt.show()
```

Image 13: show creating accuracy diagram script

```
fig = plt.figure()
plt.plot(hist.history['loss'], color='teal', label='loss')
plt.plot(hist.history['val_loss'], color='orange', label='val_loss')
fig.suptitle('Loss', fontsize=20)
plt.legend(loc="upper left")
plt.show()
```

Image 14: show creating loss diagram script

3.1.8 Evaluate model - checks the effects of training the model by testing it on the previously unused test set.

```
#Evaluate model
score = model.evaluate(test, verbose = 0)

print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Image 15: show testing model script

3.2 Data structures

In the folder of name 'Data1', there are images grouped into categories within separate folders. These data have been loaded into a structure of type '_BatchDataset'. During the division of the dataset, the type of container that stored the data changed to '_TakeDataset' as a result of the 'take' operation.

3.3 UI

The project does not include a specialized user interface. It has been built using scripts in Jupyter Lab. The code and outputs are organized in a clear and understandable manner, making the implementation of a user interface unnecessary. This approach allows for easy modification of the model configuration, which is particularly helpful when conducting experiments with different model configurations.

4. Experiments

4.1 Training few models on main dataset

4.1.1 Experiment description

The main focus of the experiments was to prepare 7 variations of a single model. These variations differed slightly from each other, allowing for an evaluation of the impact of different modifications on the network's learning outcomes. The overall configuration of the model was described in section 3.1.4. All other configurations were modified versions of this model.

The images were initially loaded at a size of 128x128. However, training the neural network on such a dataset took many hours, which significantly hindered conducting experiments. Therefore, the image size was reduced to 32x32. Interestingly, the accuracy achieved by the neural network trained on 128x128 images was only slightly higher than those trained on 32x32 images. Thus, using 128x128 images turned out to be impractical, as the cost in terms of training time and hardware resources outweighed the marginal improvement in results.

4.1.2 The prepared models are presented as follows:

	1	2	3	4	5	6	7
Con2D	3	3	3	3	2	1	3
Dropout	2x0.2	-	1x0.2	2x0.5	2x0.2	1x0.2	2x0.2
Dense	1	1	1	1	1	1	1
Flatten	1	1	1	1	1	1	1
Batch norm	3	3	3	3	2	1	-
Accuracy	0.823	0.84	0.8268	0.8093	0.8391	0.889	0.8552
Loss	0.2766	0.2618	0.2711	0.3035	0.2696	0.2301	0.2537
Time/Epoch	130s	125s	125s	130s	170s	270s	120s

Table 1: show configuration of all models

4.1.3 The training outcomes are presented below:

Model 1

```
model = Sequential()

model.add(Conv2D(64, (3, 3), padding='same', input_shape=(32, 32, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.BatchNormalization())

model.add(Conv2D(128, (3, 3), padding='same', input_shape=(32, 32, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.BatchNormalization())
model.add(Dropout(0.2))

model.add(Conv2D(256, (3, 3), padding='same', input_shape=(32, 32, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.BatchNormalization())

model.add(Flatten())
model.add(Dropout(0.2))
model.add(Dense(1024, activation='relu'))
model.add(Dense(33, activation='softmax'))
```

Image 17: show model configuration

```
Epoch 1/20
1050/1050 [=====] - 135s 127ms/step - loss: 0.9651 - accuracy: 0.6813
Epoch 2/20
1050/1050 [=====] - 133s 127ms/step - loss: 0.4304 - accuracy: 0.7822
Epoch 3/20
1050/1050 [=====] - 133s 127ms/step - loss: 0.3706 - accuracy: 0.7919
Epoch 4/20
1050/1050 [=====] - 131s 124ms/step - loss: 0.3458 - accuracy: 0.7984
Epoch 5/20
1050/1050 [=====] - 130s 124ms/step - loss: 0.3415 - accuracy: 0.8016
Epoch 6/20
1050/1050 [=====] - 130s 124ms/step - loss: 0.3216 - accuracy: 0.8059
Epoch 7/20
1050/1050 [=====] - 130s 123ms/step - loss: 0.3229 - accuracy: 0.8070
Epoch 8/20
1050/1050 [=====] - 129s 123ms/step - loss: 0.3038 - accuracy: 0.8124
Epoch 9/20
1050/1050 [=====] - 129s 123ms/step - loss: 0.3110 - accuracy: 0.8096
Epoch 10/20
1050/1050 [=====] - 131s 125ms/step - loss: 0.3032 - accuracy: 0.8137
Epoch 11/20
1050/1050 [=====] - 132s 126ms/step - loss: 0.2914 - accuracy: 0.8171
Epoch 12/20
1050/1050 [=====] - 132s 126ms/step - loss: 0.2921 - accuracy: 0.8173
Epoch 13/20
1050/1050 [=====] - 129s 123ms/step - loss: 0.2892 - accuracy: 0.8188
Epoch 14/20
1050/1050 [=====] - 126s 119ms/step - loss: 0.2902 - accuracy: 0.8170
Epoch 15/20
1050/1050 [=====] - 129s 123ms/step - loss: 0.2850 - accuracy: 0.8222
Epoch 16/20
1050/1050 [=====] - 130s 124ms/step - loss: 0.2818 - accuracy: 0.8198
Epoch 17/20
1050/1050 [=====] - 130s 123ms/step - loss: 0.2812 - accuracy: 0.8219
Epoch 18/20
1050/1050 [=====] - 126s 120ms/step - loss: 0.2796 - accuracy: 0.8206
Epoch 19/20
1050/1050 [=====] - 126s 120ms/step - loss: 0.2779 - accuracy: 0.8244
Epoch 20/20
1050/1050 [=====] - 127s 121ms/step - loss: 0.2766 - accuracy: 0.8230
```

Image 18: show effects of training process

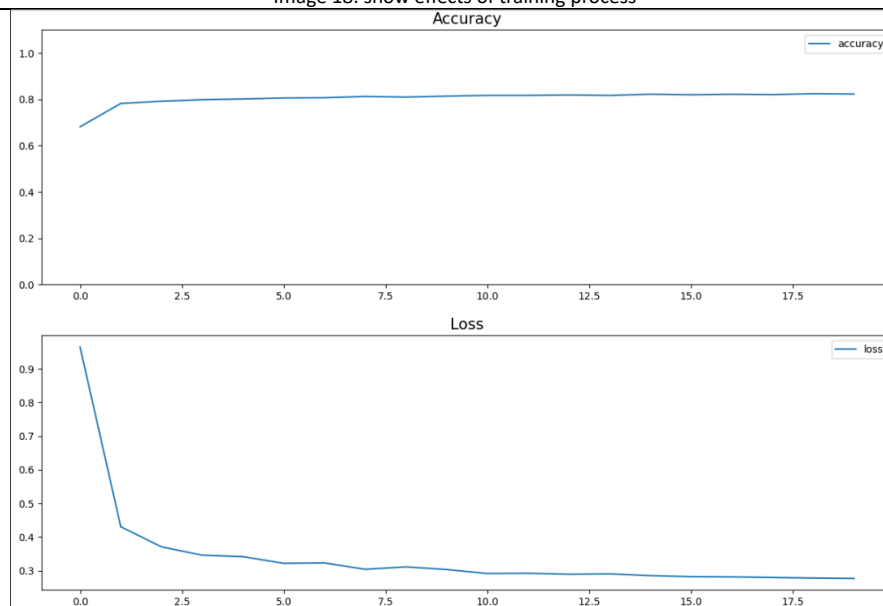


Diagram 2: show effects of training process

Model 2

```
model = Sequential()

model.add(Conv2D(64, (3, 3), padding='same', input_shape=(32, 32, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.BatchNormalization())

model.add(Conv2D(128, (3, 3), padding='same', input_shape=(32, 32, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.BatchNormalization())
#model.add(Dropout(0.2))

model.add(Conv2D(256, (3, 3), padding='same', input_shape=(32, 32, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.BatchNormalization())

model.add(Flatten())
#model.add(Dropout(0.2))
model.add(Dense(1024, activation='relu'))
model.add(Dense(33, activation='softmax'))
```

Image 19: show model configuration

```
Epoch 1/20
1050/1050 [=====] - 127s 120ms/step - loss: 0.8897 - accuracy: 0.6933
Epoch 2/20
1050/1050 [=====] - 127s 121ms/step - loss: 0.3829 - accuracy: 0.7896
Epoch 3/20
1050/1050 [=====] - 126s 120ms/step - loss: 0.3489 - accuracy: 0.7996
Epoch 4/20
1050/1050 [=====] - 122s 117ms/step - loss: 0.3233 - accuracy: 0.8045
Epoch 5/20
1050/1050 [=====] - 125s 119ms/step - loss: 0.3224 - accuracy: 0.8071
Epoch 6/20
1050/1050 [=====] - 126s 120ms/step - loss: 0.3023 - accuracy: 0.8126
Epoch 7/20
1050/1050 [=====] - 127s 121ms/step - loss: 0.3069 - accuracy: 0.8147
Epoch 8/20
1050/1050 [=====] - 127s 121ms/step - loss: 0.2894 - accuracy: 0.8196
Epoch 9/20
1050/1050 [=====] - 124s 118ms/step - loss: 0.2843 - accuracy: 0.8236
Epoch 10/20
1050/1050 [=====] - 128s 122ms/step - loss: 0.2855 - accuracy: 0.8255
Epoch 11/20
1050/1050 [=====] - 126s 120ms/step - loss: 0.2885 - accuracy: 0.8263
Epoch 12/20
1050/1050 [=====] - 122s 116ms/step - loss: 0.2773 - accuracy: 0.8268
Epoch 13/20
1050/1050 [=====] - 123s 117ms/step - loss: 0.2772 - accuracy: 0.8274
Epoch 14/20
1050/1050 [=====] - 123s 117ms/step - loss: 0.2767 - accuracy: 0.8238
Epoch 15/20
1050/1050 [=====] - 122s 116ms/step - loss: 0.2736 - accuracy: 0.8301
Epoch 16/20
1050/1050 [=====] - 123s 117ms/step - loss: 0.2783 - accuracy: 0.8251
Epoch 17/20
1050/1050 [=====] - 128s 122ms/step - loss: 0.2673 - accuracy: 0.8301
Epoch 18/20
1050/1050 [=====] - 126s 120ms/step - loss: 0.2665 - accuracy: 0.8312
Epoch 19/20
1050/1050 [=====] - 127s 120ms/step - loss: 0.2611 - accuracy: 0.8364
Epoch 20/20
1050/1050 [=====] - 127s 121ms/step - loss: 0.2618 - accuracy: 0.8401
```

Image 20: show effects of training process

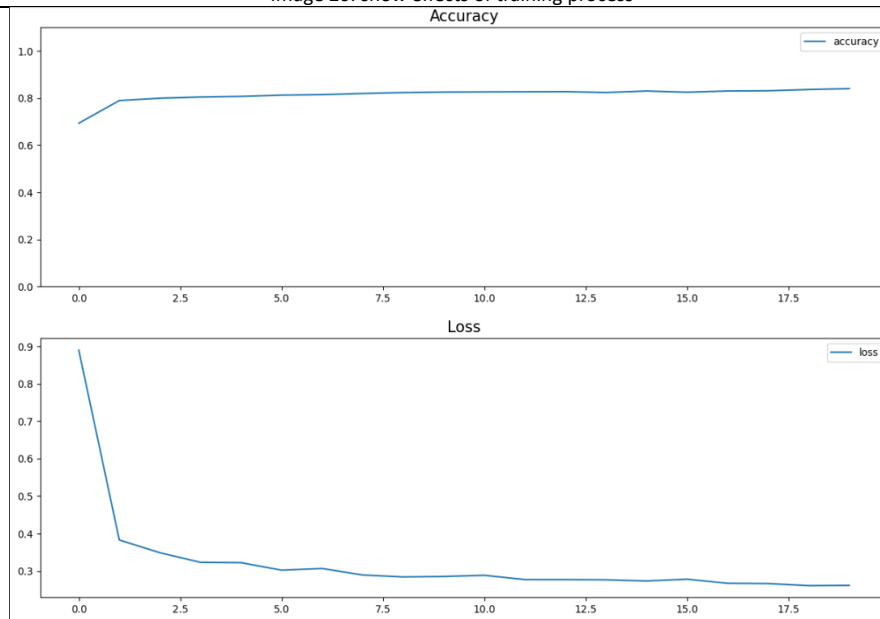


Diagram 3: show effects of training process

Model 3

```
model = Sequential()

model.add(Conv2D(64, (3, 3), padding='same', input_shape=(32, 32, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.BatchNormalization())

model.add(Conv2D(128, (3, 3), padding='same', input_shape=(32, 32, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.BatchNormalization())
#model.add(Dropout(0.2))

model.add(Conv2D(256, (3, 3), padding='same', input_shape=(32, 32, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.BatchNormalization())

model.add(Flatten())
model.add(Dropout(0.2))
model.add(Dense(1024, activation='relu'))
model.add(Dense(33, activation='softmax'))
```

Image 21: show model configuration

```
#train
logdir='logs'
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=logdir)
hist = model.fit(train, epochs=20, callbacks=[tensorboard_callback])

Epoch 1/20
1050/1050 [=====] - 129s 122ms/step - loss: 0.9013 - accuracy: 0.6920
Epoch 2/20
1050/1050 [=====] - 128s 122ms/step - loss: 0.4119 - accuracy: 0.7817
Epoch 3/20
1050/1050 [=====] - 128s 122ms/step - loss: 0.3598 - accuracy: 0.7940
Epoch 4/20
1050/1050 [=====] - 127s 121ms/step - loss: 0.3490 - accuracy: 0.7964
Epoch 5/20
1050/1050 [=====] - 126s 120ms/step - loss: 0.3299 - accuracy: 0.8019
Epoch 6/20
1050/1050 [=====] - 124s 118ms/step - loss: 0.3164 - accuracy: 0.8084
Epoch 7/20
1050/1050 [=====] - 124s 118ms/step - loss: 0.3226 - accuracy: 0.8077
Epoch 8/20
1050/1050 [=====] - 124s 118ms/step - loss: 0.3016 - accuracy: 0.8101
Epoch 9/20
1050/1050 [=====] - 124s 118ms/step - loss: 0.2926 - accuracy: 0.8161
Epoch 10/20
1050/1050 [=====] - 124s 118ms/step - loss: 0.3165 - accuracy: 0.8103
Epoch 11/20
1050/1050 [=====] - 123s 118ms/step - loss: 0.2885 - accuracy: 0.8172
Epoch 12/20
1050/1050 [=====] - 123s 117ms/step - loss: 0.2868 - accuracy: 0.8178
Epoch 13/20
1050/1050 [=====] - 124s 118ms/step - loss: 0.2850 - accuracy: 0.8221
Epoch 14/20
1050/1050 [=====] - 124s 118ms/step - loss: 0.2810 - accuracy: 0.8260
Epoch 15/20
1050/1050 [=====] - 124s 118ms/step - loss: 0.2826 - accuracy: 0.8235
Epoch 16/20
1050/1050 [=====] - 128s 122ms/step - loss: 0.2815 - accuracy: 0.8241
Epoch 17/20
1050/1050 [=====] - 129s 123ms/step - loss: 0.2717 - accuracy: 0.8276
Epoch 18/20
1050/1050 [=====] - 127s 121ms/step - loss: 0.2792 - accuracy: 0.8266
Epoch 19/20
1050/1050 [=====] - 124s 118ms/step - loss: 0.2780 - accuracy: 0.8271
Epoch 20/20
1050/1050 [=====] - 124s 118ms/step - loss: 0.2721 - accuracy: 0.8268
```

Image 22: show effects of training process

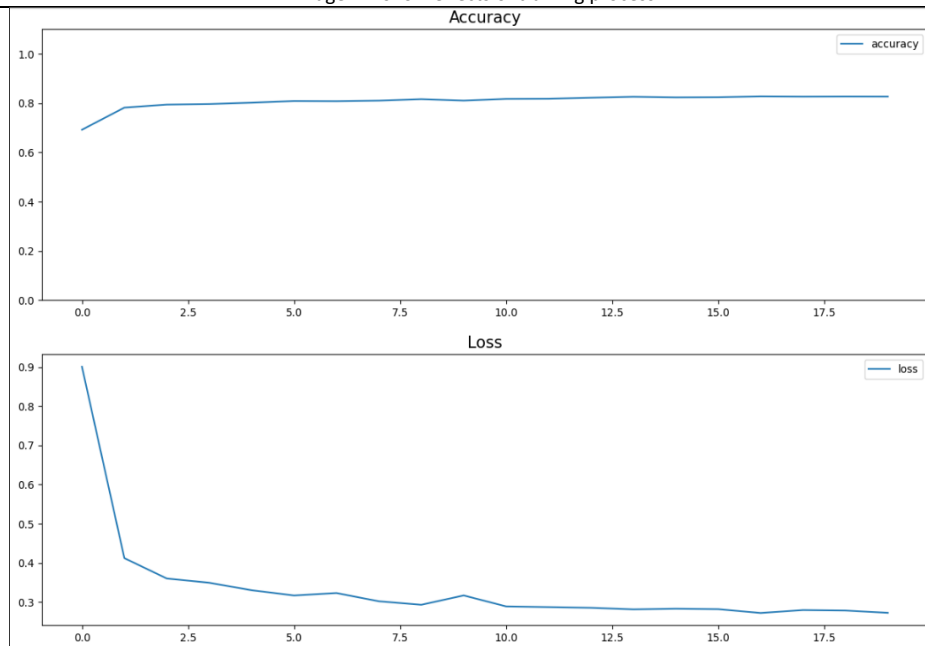


Diagram 4: show effects of training process

Model 4

```
model = Sequential()

model.add(Conv2D(64, (3, 3), padding='same', input_shape=(32, 32, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.BatchNormalization())

model.add(Conv2D(128, (3, 3), padding='same', input_shape=(32, 32, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.BatchNormalization())
model.add(Dropout(0.5))
model.add(Dropout(0.2))
model.add(Conv2D(256, (3, 3), padding='same', input_shape=(32, 32, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.BatchNormalization())

model.add(Flatten())
model.add(Dropout(0.5))
model.add(Dropout(0.2))
model.add(Dense(1024, activation='relu'))
model.add(Dense(33, activation='softmax'))
```

Image 23: show model configuration

```
hist = model.fit(train, epochs=20, callbacks=[tensorboard_callback])

Epoch 1/20
1050/1050 [=====] - 129s 122ms/step - loss: 1.1928 - accuracy: 0.6296
Epoch 2/20
1050/1050 [=====] - 129s 123ms/step - loss: 0.5548 - accuracy: 0.7517
Epoch 3/20
1050/1050 [=====] - 131s 124ms/step - loss: 0.4749 - accuracy: 0.7720
Epoch 4/20
1050/1050 [=====] - 127s 121ms/step - loss: 0.4339 - accuracy: 0.7780
Epoch 5/20
1050/1050 [=====] - 124s 118ms/step - loss: 0.4151 - accuracy: 0.7816
Epoch 6/20
1050/1050 [=====] - 128s 122ms/step - loss: 0.3892 - accuracy: 0.7908
Epoch 7/20
1050/1050 [=====] - 128s 122ms/step - loss: 0.3809 - accuracy: 0.7904
Epoch 8/20
1050/1050 [=====] - 128s 122ms/step - loss: 0.3560 - accuracy: 0.7971
Epoch 9/20
1050/1050 [=====] - 127s 121ms/step - loss: 0.3512 - accuracy: 0.7959
Epoch 10/20
1050/1050 [=====] - 125s 119ms/step - loss: 0.3449 - accuracy: 0.7971
Epoch 11/20
1050/1050 [=====] - 129s 123ms/step - loss: 0.3412 - accuracy: 0.7988
Epoch 12/20
1050/1050 [=====] - 128s 122ms/step - loss: 0.3315 - accuracy: 0.8015
Epoch 13/20
1050/1050 [=====] - 129s 123ms/step - loss: 0.3261 - accuracy: 0.8021
Epoch 14/20
1050/1050 [=====] - 129s 122ms/step - loss: 0.3211 - accuracy: 0.8026
Epoch 15/20
1050/1050 [=====] - 129s 122ms/step - loss: 0.3183 - accuracy: 0.8066
Epoch 16/20
1050/1050 [=====] - 129s 123ms/step - loss: 0.3154 - accuracy: 0.8055
Epoch 17/20
1050/1050 [=====] - 131s 125ms/step - loss: 0.3093 - accuracy: 0.8055
Epoch 18/20
1050/1050 [=====] - 130s 124ms/step - loss: 0.3103 - accuracy: 0.8083
Epoch 19/20
1050/1050 [=====] - 129s 123ms/step - loss: 0.3043 - accuracy: 0.8095
Epoch 20/20
1050/1050 [=====] - 130s 123ms/step - loss: 0.3035 - accuracy: 0.8093
```

Image 24: show effects of training process

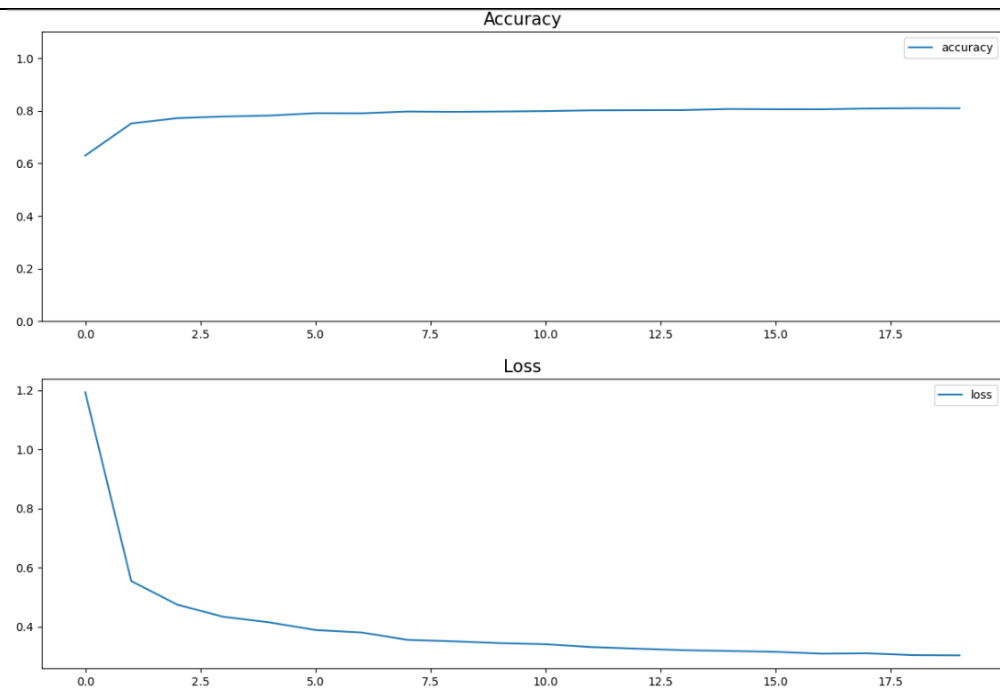


Diagram 5: show effects of training process

Model 5

```
model = Sequential()

model.add(Conv2D(64, (3, 3), padding='same', input_shape=(32, 32, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.BatchNormalization())

model.add(Conv2D(128, (3, 3), padding='same', input_shape=(32, 32, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.BatchNormalization())
#model.add(Dropout(0.5))
model.add(Dropout(0.2))

#model.add(Conv2D(256, (3, 3), padding='same', input_shape=(32, 32, 3), activation='relu'))
#model.add(MaxPooling2D(pool_size=(2, 2)))
#model.add(tf.keras.layers.BatchNormalization())

model.add(Flatten())
#model.add(Dropout(0.5))
model.add(Dropout(0.2))
model.add(Dense(1024, activation='relu'))
model.add(Dense(32, activation='softmax'))
```

Image 25: show model configuration

```
#train
logdir='logs'
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=logdir)
hist = model.fit(train, epochs=20, callbacks=[tensorboard_callback])

Epoch 1/20
1050/1050 [=====] - 173s 164ms/step - loss: 1.1553 - accuracy: 0.6549
Epoch 2/20
1050/1050 [=====] - 169s 161ms/step - loss: 0.4688 - accuracy: 0.7718
Epoch 3/20
1050/1050 [=====] - 167s 159ms/step - loss: 0.3876 - accuracy: 0.7912
Epoch 4/20
1050/1050 [=====] - 167s 159ms/step - loss: 0.3669 - accuracy: 0.8019
Epoch 5/20
1050/1050 [=====] - 167s 159ms/step - loss: 0.3467 - accuracy: 0.8030
Epoch 6/20
1050/1050 [=====] - 169s 161ms/step - loss: 0.3270 - accuracy: 0.8110
Epoch 7/20
1050/1050 [=====] - 170s 162ms/step - loss: 0.3197 - accuracy: 0.8138
Epoch 8/20
1050/1050 [=====] - 169s 160ms/step - loss: 0.3291 - accuracy: 0.8115
Epoch 9/20
1050/1050 [=====] - 166s 158ms/step - loss: 0.3064 - accuracy: 0.8174
Epoch 10/20
1050/1050 [=====] - 166s 158ms/step - loss: 0.3041 - accuracy: 0.8174
Epoch 11/20
1050/1050 [=====] - 166s 158ms/step - loss: 0.2976 - accuracy: 0.8229
Epoch 12/20
1050/1050 [=====] - 167s 159ms/step - loss: 0.2912 - accuracy: 0.8243
Epoch 13/20
1050/1050 [=====] - 166s 158ms/step - loss: 0.2866 - accuracy: 0.8281
Epoch 14/20
1050/1050 [=====] - 169s 161ms/step - loss: 0.2885 - accuracy: 0.8262
Epoch 15/20
1050/1050 [=====] - 170s 162ms/step - loss: 0.2837 - accuracy: 0.8267
Epoch 16/20
1050/1050 [=====] - 170s 162ms/step - loss: 0.2805 - accuracy: 0.8295
Epoch 17/20
1050/1050 [=====] - 167s 159ms/step - loss: 0.2734 - accuracy: 0.8349
Epoch 18/20
1050/1050 [=====] - 170s 162ms/step - loss: 0.2764 - accuracy: 0.8336
Epoch 19/20
1050/1050 [=====] - 171s 163ms/step - loss: 0.2714 - accuracy: 0.8373
Epoch 20/20
1050/1050 [=====] - 172s 163ms/step - loss: 0.2696 - accuracy: 0.8391
```

Image 26: show effects of training process

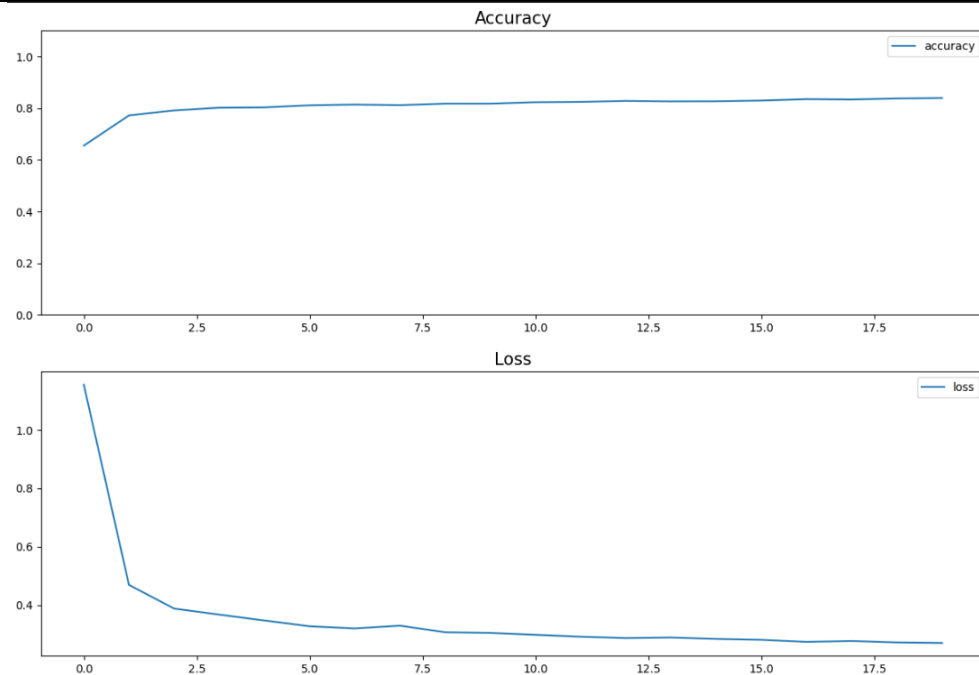


Diagram 6: show effects of training process

Model 6

```
model = Sequential()

model.add(Conv2D(64, (3, 3), padding='same', input_shape=(32, 32, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.BatchNormalization())

model.add(Conv2D(128, (3, 3), padding='same', input_shape=(32, 32, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.BatchNormalization())
#model.add(Dropout(0.5))
model.add(Dropout(0.2))

#model.add(Conv2D(256, (3, 3), padding='same', input_shape=(32, 32, 3), activation='relu'))
#model.add(MaxPooling2D(pool_size=(2, 2)))
#model.add(tf.keras.layers.BatchNormalization())

model.add(Flatten())
#model.add(Dropout(0.5))
model.add(Dropout(0.2))
model.add(Dense(1024, activation='relu'))
model.add(Dense(33, activation='softmax'))
```

Image 27: show model configuration

```
logdir='logs'
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=logdir)
hist = model.fit(train, epochs=20, callbacks=[tensorboard_callback])
```

```
Epoch 1/20
1850/1850 [=====] - 271s 258ms/step - loss: 1.1682 - accuracy: 0.6290
Epoch 2/20
1850/1850 [=====] - 270s 257ms/step - loss: 0.5222 - accuracy: 0.7592
Epoch 3/20
1850/1850 [=====] - 269s 257ms/step - loss: 0.4028 - accuracy: 0.7904
Epoch 4/20
1850/1850 [=====] - 268s 255ms/step - loss: 0.3858 - accuracy: 0.7952
Epoch 5/20
1850/1850 [=====] - 271s 258ms/step - loss: 0.3545 - accuracy: 0.8043
Epoch 6/20
1850/1850 [=====] - 270s 257ms/step - loss: 0.3394 - accuracy: 0.8102
Epoch 7/20
1850/1850 [=====] - 272s 259ms/step - loss: 0.3371 - accuracy: 0.8119
Epoch 8/20
1850/1850 [=====] - 273s 260ms/step - loss: 0.3217 - accuracy: 0.8225
Epoch 9/20
1850/1850 [=====] - 271s 258ms/step - loss: 0.3210 - accuracy: 0.8232
Epoch 10/20
1850/1850 [=====] - 268s 255ms/step - loss: 0.3158 - accuracy: 0.8262
Epoch 11/20
1850/1850 [=====] - 267s 254ms/step - loss: 0.3059 - accuracy: 0.8326
Epoch 12/20
1850/1850 [=====] - 268s 255ms/step - loss: 0.2972 - accuracy: 0.8364
Epoch 13/20
1850/1850 [=====] - 266s 253ms/step - loss: 0.2887 - accuracy: 0.8446
Epoch 14/20
1850/1850 [=====] - 266s 253ms/step - loss: 0.2765 - accuracy: 0.8523
Epoch 15/20
1850/1850 [=====] - 267s 254ms/step - loss: 0.2672 - accuracy: 0.8601
Epoch 16/20
1850/1850 [=====] - 267s 255ms/step - loss: 0.2592 - accuracy: 0.8684
Epoch 17/20
1850/1850 [=====] - 268s 255ms/step - loss: 0.2520 - accuracy: 0.8745
Epoch 18/20
1850/1850 [=====] - 269s 256ms/step - loss: 0.2401 - accuracy: 0.8807
Epoch 19/20
1850/1850 [=====] - 273s 260ms/step - loss: 0.2301 - accuracy: 0.8890
Epoch 20/20
1850/1850 [=====] - 272s 259ms/step - loss: 0.2183 - accuracy: 0.8959
```

Image 28: show effects of training process

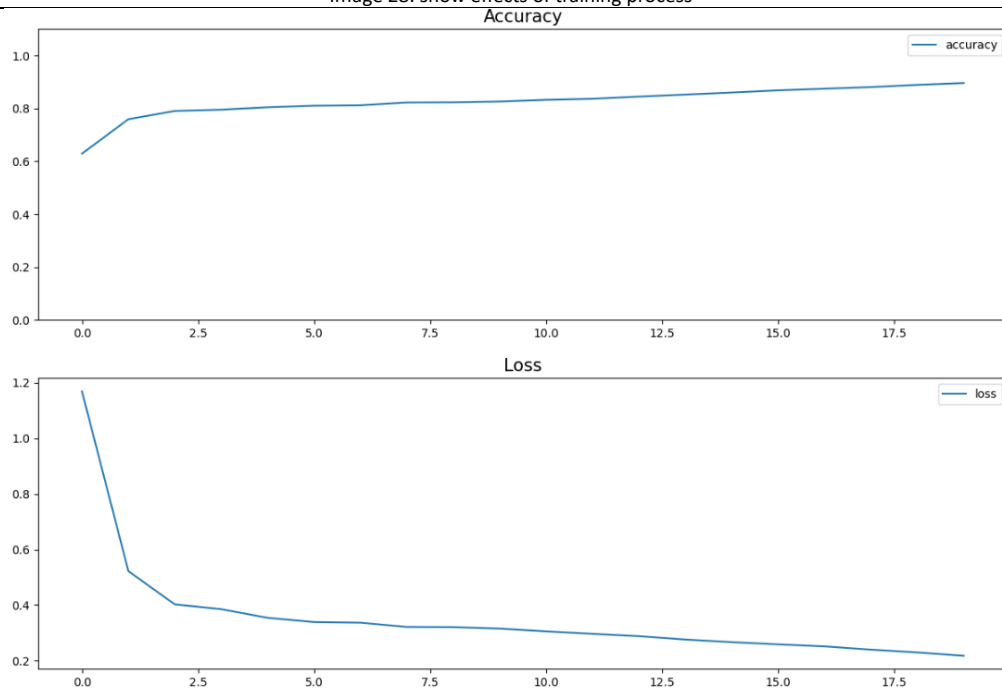


Diagram 7: show effects of training process

Model 7

```
model = Sequential()

model.add(Conv2D(64, (3, 3), padding='same', input_shape=(32, 32, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
#model.add(tf.keras.layers.BatchNormalization())

model.add(Conv2D(128, (3, 3), padding='same', input_shape=(32, 32, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
#model.add(tf.keras.layers.BatchNormalization())
#model.add(Dropout(0.5))
model.add(Dropout(0.2))

model.add(Conv2D(256, (3, 3), padding='same', input_shape=(32, 32, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
#model.add(tf.keras.layers.BatchNormalization())

model.add(Flatten())
#model.add(Dropout(0.5))
model.add(Dropout(0.2))
model.add(Dense(1024, activation='relu'))
model.add(Dense(33, activation='softmax'))
```

Image 29: show model configuration

```
from IPython.display import clear_output
logdir='logs'
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=logdir)
hist = model.fit(train, epochs=20, callbacks=[tensorboard_callback])
```

Epoch	Time	Loss	Accuracy
Epoch 1/20			
1050/1050	132s 125ms/step	loss: 1.1330	accuracy: 0.6175
Epoch 2/20			
1050/1050	124s 118ms/step	loss: 0.4587	accuracy: 0.7719
Epoch 3/20			
1050/1050	126s 120ms/step	loss: 0.3624	accuracy: 0.7944
Epoch 4/20			
1050/1050	126s 120ms/step	loss: 0.3349	accuracy: 0.7966
Epoch 5/20			
1050/1050	122s 117ms/step	loss: 0.3201	accuracy: 0.8043
Epoch 6/20			
1050/1050	123s 117ms/step	loss: 0.3138	accuracy: 0.8055
Epoch 7/20			
1050/1050	121s 115ms/step	loss: 0.3041	accuracy: 0.8104
Epoch 8/20			
1050/1050	123s 117ms/step	loss: 0.3031	accuracy: 0.8101
Epoch 9/20			
1050/1050	122s 117ms/step	loss: 0.3010	accuracy: 0.8114
Epoch 10/20			
1050/1050	121s 116ms/step	loss: 0.2980	accuracy: 0.8135
Epoch 11/20			
1050/1050	119s 113ms/step	loss: 0.2945	accuracy: 0.8145
Epoch 12/20			
1050/1050	119s 114ms/step	loss: 0.2889	accuracy: 0.8187
Epoch 13/20			
1050/1050	120s 114ms/step	loss: 0.2875	accuracy: 0.8213
Epoch 14/20			
1050/1050	120s 114ms/step	loss: 0.2891	accuracy: 0.8226
Epoch 15/20			
1050/1050	120s 114ms/step	loss: 0.2767	accuracy: 0.8301
Epoch 16/20			
1050/1050	120s 114ms/step	loss: 0.2759	accuracy: 0.8316
Epoch 17/20			
1050/1050	120s 115ms/step	loss: 0.2774	accuracy: 0.8350
Epoch 18/20			
1050/1050	120s 114ms/step	loss: 0.2688	accuracy: 0.8414
Epoch 19/20			
1050/1050	120s 114ms/step	loss: 0.2617	accuracy: 0.8484
Epoch 20/20			
1050/1050	120s 114ms/step	loss: 0.2537	accuracy: 0.8552

Image 30: show effects of training process

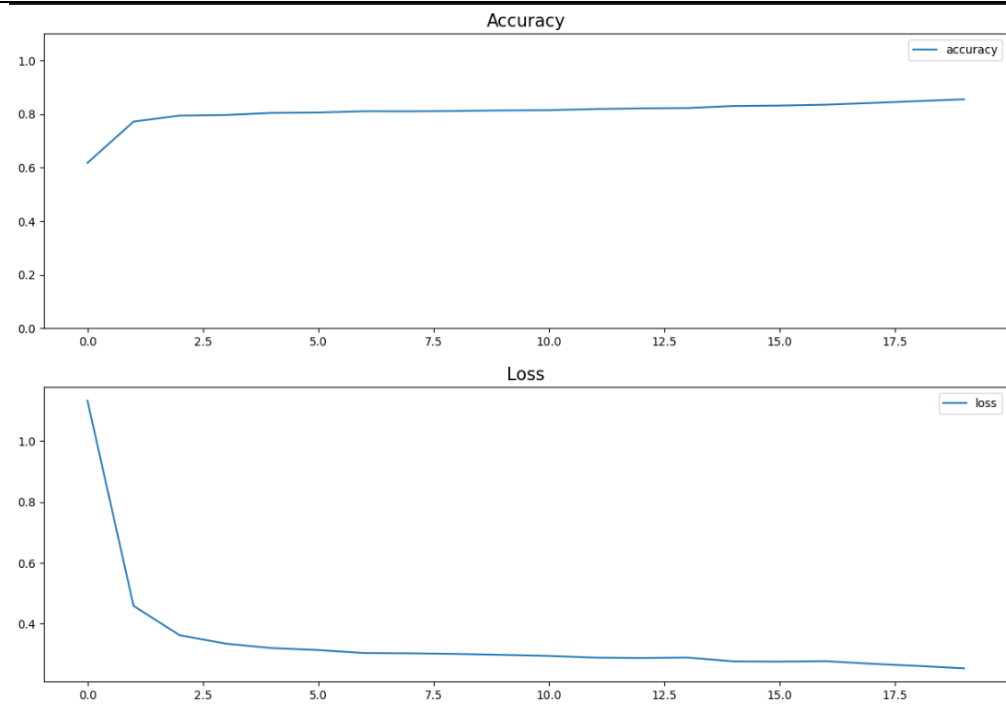


Diagram 8: show effects of training process

4.1.4 Training effects:

Model number	Accuracy	Loss	Comment
1	0.823	0.2766	
2	0.84	0.2618	The absence of a dropout layer resulted in an increase in accuracy and a decrease in loss compared to Model 1. This effect was not anticipated before training.
3	0.8268	0.2711	The removal of one dropout layer slightly reduced both the accuracy and loss, which was quite unexpected, especially after observing the training results of Model 2. However, the differences are minimal enough that they may be attributed to random chance.
4	0.8093	0.3035	Increasing the values for both dropout layers resulted in a decrease in accuracy and an increase in loss. The differences in effects are greater than those described in the case of Model 3, and they are not small enough to be ignored.
5	0.8391	0.2696	The removal of one Conv2D layer resulted in a significant increase in accuracy, although smaller than in the case of Model 2. However, the loss only decreased to a small extent. Interestingly, the training time for the model significantly increased from approximately 125 seconds to 170 seconds.
6	0.889	0.2301	In this model, the accuracy increased significantly to 0.889, while the loss decreased to 0.23. This represents a substantial improvement compared to each of the previous models. However, the training time increased dramatically.
7	0.8552	0.2537	There was a considerable improvement in the training results, along with a very slight decrease in training time.

Table 2: present effects of training models

4.1.5 Testing models:

Model	Accuracy	Loss
1	0.7837	0.386
2	0.796	0.364
3	0.7987	0.356
4	0.808	0.291
5	0.6839	0.805
6	0.780	0.769
7	0.811	0.335

Table 3: presents effects of testing models

4.1.6 Conclusions from the experiment:

The decrease in accuracy and increase in loss during the testing phase were expected, but the drastic differences observed in the case of Models 5 and 6 were a big surprise. Model 6 appeared to be the most effective variant during the training phase, but it performed worse than the baseline model during testing. Models 7 and 4 turned out to be the most effective models.

Layer	Comment
Con2D	Reducing the number of convolutional layers during the training phase appeared to result in an apparent increase in accuracy. However,

	when testing models with fewer layers, a significant decrease in accuracy and an increase in training time were observed.
BatchNormalization	The Batch Normalization layer was added to improve the training results of the neural network and reduce the training time. However, Model 7, which was the only one without this layer, achieved the best result, and its training time was the shortest.
Dropout	In the case of the trained models, using a dropout layer with a parameter of 0.2 resulted in a decrease in accuracy, as observed in Models 1, 2, and 3. However, Model 4, which had the largest dropout layer, exhibited higher accuracy than the previous three models.

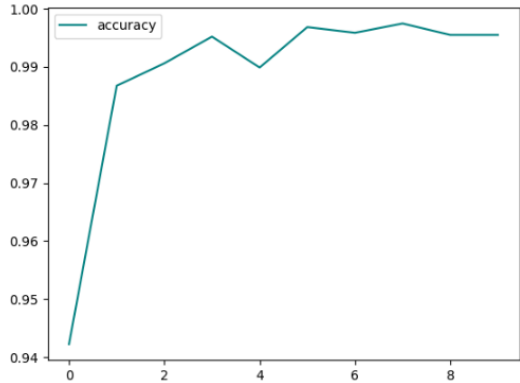
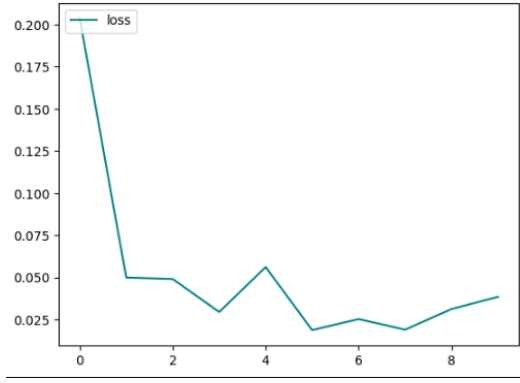
Table 4: presents conclusions about impact layers on training effects

4.2 Experiment with MNIST dataset

4.2.1 Experiment description

To examine the impact of the dataset on the training effect of the neural network, the previously prepared model configurations were retrained on a different dataset.

4.2.2 Training effects

Model	Accuracy	Loss	Diagrams
1	0.9954	0.0385	 <p>Diagram 9: presents accuracy effects of training process</p>  <p>Diagram 10: presents loss effects of training process</p>

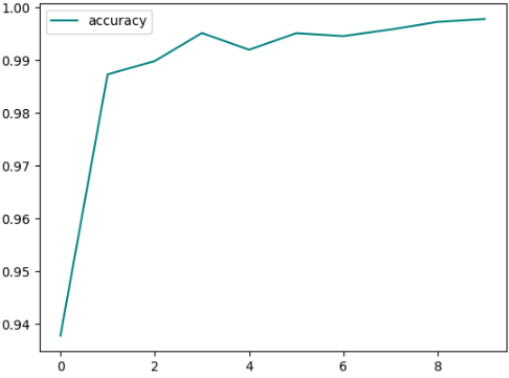
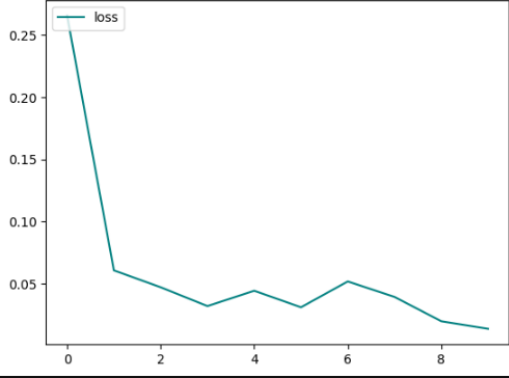
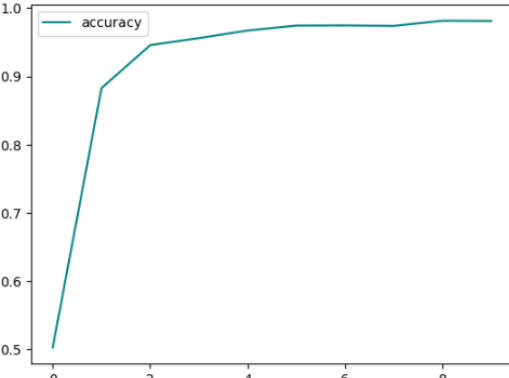
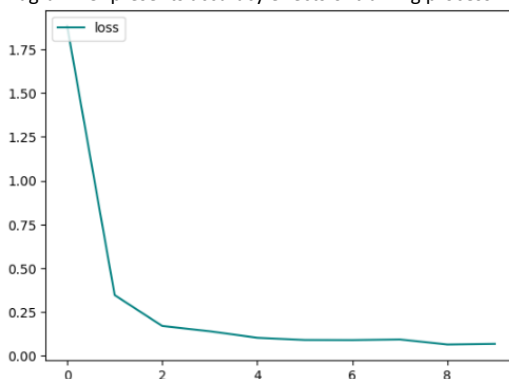
6	0.9977	0.0137	 <p>Diagram 11: presents accuracy effects of training process</p>  <p>Diagram 12: presents loss effects of training process</p>
7	0.9809	0.0673	 <p>Diagram 13: presents accuracy effects of training process</p>  <p>Diagram 14: presents loss effects of training process</p>

Table 5: presents effect of training models on mnist data set

4.2.3 Testing effects:

Model	Accuracy	Loss
1	0.935	0.727
6	0.8997	1.6249
7	0.9358	0.33

Table 6: presents effects of testing models on mnist data set

4.2.4 Results:

In the case of models 1 and 6, a significant increase in accuracy resulted in an increase in loss as well. The testing results for model 7 show a substantial increase in accuracy and the same loss score as when training that model on the main dataset.

5. Conclusions

During the training of neural networks, two factors should be taken into account: the quality of the dataset and a properly prepared and configured model. These two factors have the most significant impact on the final performance of the network.

The main challenge encountered during the network development was the conflict between the tools (PyTorch, TensorFlow) and the CUDA drivers of the graphics card used for training. This significantly prolonged the training process and limited the possibilities for further experimentation.

The utilization of artificial intelligence can greatly facilitate communication with deaf and hard-of-hearing individuals by creating a sign language translator that can be accessible through mobile phones, or by developing efficient tools for learning sign language.

6. Future work

The project can be expanded in three ways:

- Adding new signs for recognition (such as "thank you," "I love you," etc.). One major limitation in expanding the project in this direction is the limited number of available datasets. The amount of information that can be conveyed through American Sign Language is vast, making it challenging to gather a sufficient amount of data to properly train a neural network without collaboration with a research institution.
- Adding sign recognition from video files - this would allow for an expansion of the recognized gestures. However, incorporating this functionality into the project may be challenging due to the choice of using Image Classification instead of Pose Estimation methods.
- Adding a module for real-time sign recognition - this would require connecting a camera and utilizing appropriate libraries for capturing images from the camera and locating the person performing the gestures in real-time.

7. Sources

<https://www.kaggle.com/datasets/datamunge/sign-language-mnist>

<https://www.kaggle.com/datasets/kapillondhe/american-sign-language>

<https://www.kaggle.com/datasets/ardamavi/27-class-sign-language-dataset>

<https://www.kaggle.com/datasets/grassknoted/asl-alphabet>

<https://www.kaggle.com/datasets/ardamavi/sign-language-digits-dataset>

<https://www.kaggle.com/datasets/muhammadkhalid/sign-language-for-numbers/code>

<https://www.kaggle.com/datasets/muhammadkhalid/sign-language-for-alphabets>

<https://keras.io>

https://www.tensorflow.org/?gad=1&gclid=Cj0KCQjwj_ajBhCqARIsAA37s0yg49ORkJ73tXufk3xfpMjLsQGsW2pRRrC96vjwIC3ICIA3KqH8_hYaAgGiEALw_wcB&hl=pl

8. Links to project:

<https://github.com/maciekcaban/ASL-Recognition-with-python-/tree/main>

[https://polslpl-](https://polslpl-my.sharepoint.com/:f/g/personal/macicab865_student_polsl_pl/Ek6X4pm6hjZHpGy_yEO5QDMBI2PBunlwhXdoCdKXE3GBcw?e=oU8B4N)

[my.sharepoint.com/:f/g/personal/macicab865_student_polsl_pl/Ek6X4pm6hjZHpGy_yEO5QDMBI2PBunlwhXdoCdKXE3GBcw?e=oU8B4N](https://polslpl-my.sharepoint.com/:f/g/personal/macicab865_student_polsl_pl/Ek6X4pm6hjZHpGy_yEO5QDMBI2PBunlwhXdoCdKXE3GBcw?e=oU8B4N)