Course:

# Machine Learning Lab

Final project. Deep learning.

**Traffic Sign Recognition project**

Name: Evgenii Vylegzhanin

Date: 24.01.2022

## Index

## 1. Problem description

We decided to work with a traffic sign recognition problem. Traffic signs and symbols are easy means of communication that let the road users remain aware of basic traffic laws, along with giving warnings and information about locations and nearby amenities. They are very important for providing safety on the roads.

Traffic sign detection and recognition has gained importance with advances in image processing due to the benefits that such a system can provide. The recent developments and interest in self-driving cars have also increased the interest in this field. The expected result –reducing human errors and the number of accidents in order to increase drivers' safety and save the monetary values associated with car accidents [1].

Main traffic sign recognition applications:

- Autonomous cars
- Driver assistance systems (instantly assists drivers or automatic driving systems in detecting and recognizing traffic signs effectively)

There are lots of different types of traffic signs like speed limits, no entry, traffic signals, turn left or right, children crossing, no passing of heavy vehicles, and others.

The main objective of the project is to implement different neural networks (NN) for problem resolution such as feedforward NN, convolutional NN (CNN), and recurrent NN (RNN). Also, it would be great to check the effect of batch normalization and dropout on the performance of NNs and implement experiments with transfer learning and fine-tuning. We can vary several parameters (layers of NNs, number of epochs, batch size, train/test split proportion activation functions, loss functions). As a metric to check the performance of a NN, we will use "accuracy".

The problem of the project is related to the multi-class classification problem type.

## 2. Experimental framework

### 2.1. Dataset description

We will use a public dataset available at Kaggle "GTSRB – German Traffic Sign Recognition Benchmark" [2].

The German Traffic Sign Benchmark is a multi-class, single-image classification challenge held at the International Joint Conference on Neural Networks (IJCNN) 2011. The benchmark has the following properties:

- Single-image, multi-class classification problem
- More than 40 classes
- More than 50 000 images in total
- Large, lifelike database

The dataset contains several folders. The first one is 'Train'. It has 43 folders each representing a different class. The 'Test' folder has images that we will use to evaluate our model performance.

### 2.2. Experimental setup

We will use train-test-split to split our train dataset into two parts 80% for training and 20% for the validation stage. Probably, we will try different proportions as well (70/30 or 60/40).

So, we have training and validation sets to train our NNs and test dataset from a different file to evaluate the final performance of the model.

### 2.3. Basics of the exploratory data analysis and preprocessing

In order to manage the data more easily, we have to prepare the convenient form of the data. Let's download our datasets and create a .ipynb file, then open Jupyter notebook and write a script to transform all images into several NumPy arrays.

The first step is to install all necessary libraries and import them.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
from PIL import Image
import os
from sklearn.model_selection import train_test_split
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPool2D, Dense, Flatte
n, Dropout
```

The second step is to iterate over all the classes and append images and their labels in the data and labels list. After that, we convert lists into NumPy arrays.

```
data = []
labels = []
classes = 43
cur_path = os.getcwd()
for i in range(classes):
  path = os.path.join(cur_path, 'train', str(i))
  images = os.listdir(path)
  for a in images:
    try:
      image = Image.open(path + '\\' + a)
      image = image.resize((30, 30))
      image = np.array(image)
      #sim = Image.fromarray(image)
      data.append(image)
      labels.append(i)
    except:
      print('Error loading image')
data = np.array(data)
labels = np.array(labels)
```

As a result, we have a data array of shape (39209, 30, 30, 3) which means that there are 39209 images of size (30x30) pixels and 3 colors (RGB value).

The next step is to divide our train data into test and validation arrays and convert labels into categories:

```
X_train, X_val, y_train, y_val = train_test_split(data, labels, test
_size=0.2, random_state=42)
y_train = to_categorical(y_train, 43)
y_val = to_categorical(y_val, 43)
```

The last part is to perform the same steps for the test file and save all NumPy arrays into files which we will use for further work in Google Colab.

```
y_test = pd.read_csv('Test.csv')
labels = y_test['ClassId'].values
imgs = y_test["Path"].values
data = []
for img in imgs:
    image = Image.open(img)
    image = image.resize((30,30))
    data.append(np.array(image))
X_test = np.array(data)
np.save('X_train.npy', X_train)
np.save('X_val.npy', X_val)
np.save('y_train.npy', y_train)
np.save('y_val.npy', y_val)
np.save('X_test.npy', X_test)
```

3. Baseline solutions for different neural network types

## 3.1. Convolutional neural network

The first step here is to create a CNN. It is said that for image classification purposes CNNs are the best choice.

Initialization:

```
epochs = 20
batch_size = 64
loss_function = 'categorical_crossentropy'
optimizer = 'adam'
metrics = ['accuracy']
```

Adding some layers and compiling the model:

```
model = Sequential()
model.add(Conv2D(filters=32, kernel_size=(5,5), activation='relu', i
nput_shape=X_train.shape[1:]))
model.add(Conv2D(filters=32, kernel_size=(5,5), activation='relu'))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Dropout(rate=0.25))
model.add(Conv2D(filters=32, kernel_size=(3,3), activation='relu'))
model.add(Conv2D(filters=32, kernel_size=(3,3), activation='relu'))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Dropout(rate=0.25))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(rate=0.5))
model.add(Dense(43, activation='softmax'))
model.compile(loss=loss_function, optimizer=optimizer, metrics=metri
cs)
```

Then let's train the CNN model.

```
history = model.fit(X_train, y_train, batch_size=batch_size, epochs=
epochs, validation_data=(X_val, y_val))
```

The last part is to plot the graphs for accuracy and loss of training and validation datasets.

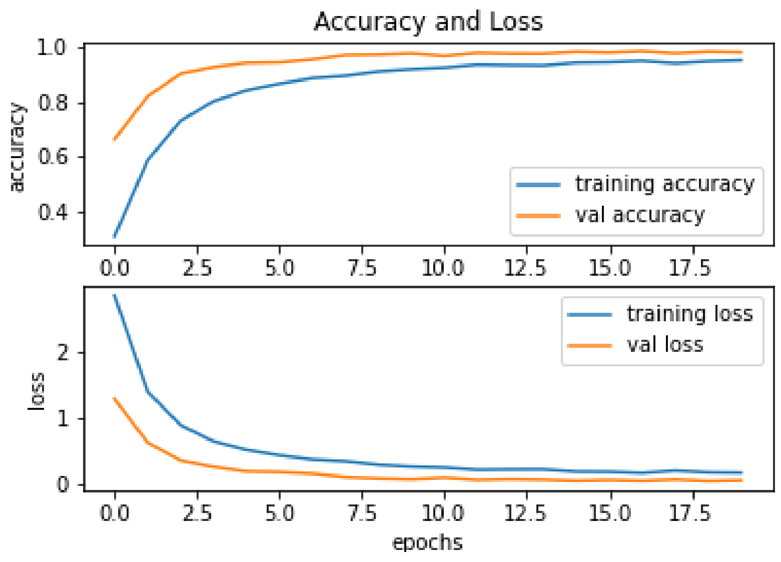


Figure 1. Accuracy and loss of the CNN baseline solution

And finally, the accuracy on the test dataset.

```
predict_x=model.predict(X_test)
classes_x=np.argmax(predict_x,axis=1)
labels = y_test['ClassId'].values
print(accuracy_score(labels, classes_x))
```

We got accuracy equaled to 94%.

## 3.2. Feed-forward neural network

Let’s initialize some parameters:

```
epochs = 100
batch_size = 64
loss_function = 'categorical_crossentropy'
metrics = ['accuracy']
```

Then in order to work with fully connected neural networks, we have to reshape datasets.

```
X_train = X_train.reshape([-
1, X_train.shape[1]*X_train.shape[2]*X_train.shape[3]])
X_test  = X_test.reshape([-
1, X_test.shape[1]*X_test.shape[2]*X_test.shape[3]])
X_val = X_val.reshape([-
1, X_val.shape[1]*X_val.shape[2]*X_val.shape[3]])
```

The next step is to create and compile a feed-forward neural network.

```
model = Sequential()
model.add(Dense(256, input_dim=X_train.shape[1], activation="relu"))
model.add(Dense(128, activation='sigmoid'))
model.add(Dense(64, activation='sigmoid'))
model.add(Dense(43, activation='softmax'))
sgd = SGD(lr=0.0005, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss=loss_function, optimizer=sgd, metrics=metrics)
```

Let’s fit it and make plot some graphs.

```
history = model.fit(X_train, y_train, batch_size=batch_size, epochs=
epochs, validation_data=(X_val, y_val))
```

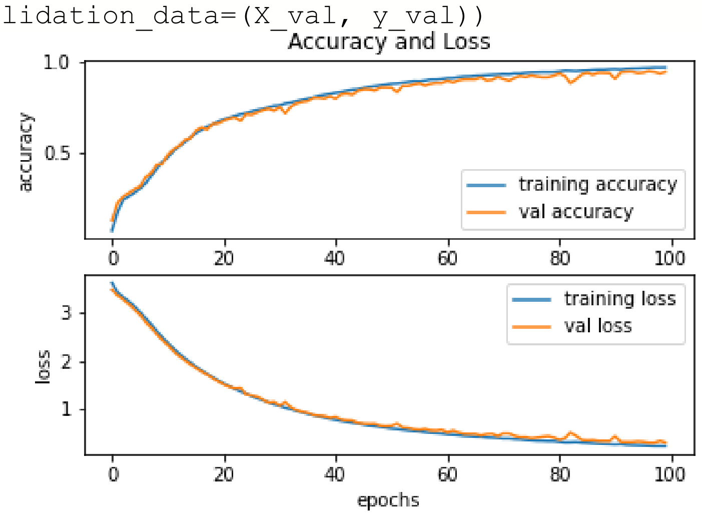


Figure 2. Accuracy and loss of feed-forward neural network

The final accuracy on the test dataset is 86%.

## 3.3. Recurrent neural network

Again, let’s initialize some parameters:

```
epochs = 7
batch_size = 64
loss_func = 'categorical_crossentropy'
metrics = ['accuracy']
opt = tf.keras.optimizers.Adam(lr=1e-3, decay=1e-5)
```

Then build, compile and fit the model. Here our model is created with the combination of CNN and LSTM (long short-term memory – an artificial recurrent neural network architecture).

```
model = Sequential()
model.add(Conv2D(filters=32, kernel_size=(5,5), activation='relu',
                                  input_shape=X_train.shape[1:]))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Lambda(ReshapeLayer))
model.add(LSTM(128, return_sequences=True, activation='relu'))
model.add(Dropout(0.2))
model.add(LSTM(128, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(43, activation='softmax'))
model.compile(loss=loss_func, optimizer=opt, metrics=metrics)
history = model.fit(X_train, y_train, batch_size=batch_size, epochs=
epochs, validation_data=(X_val, y_val))
```

Finally, let’s plot accuracy and loss.

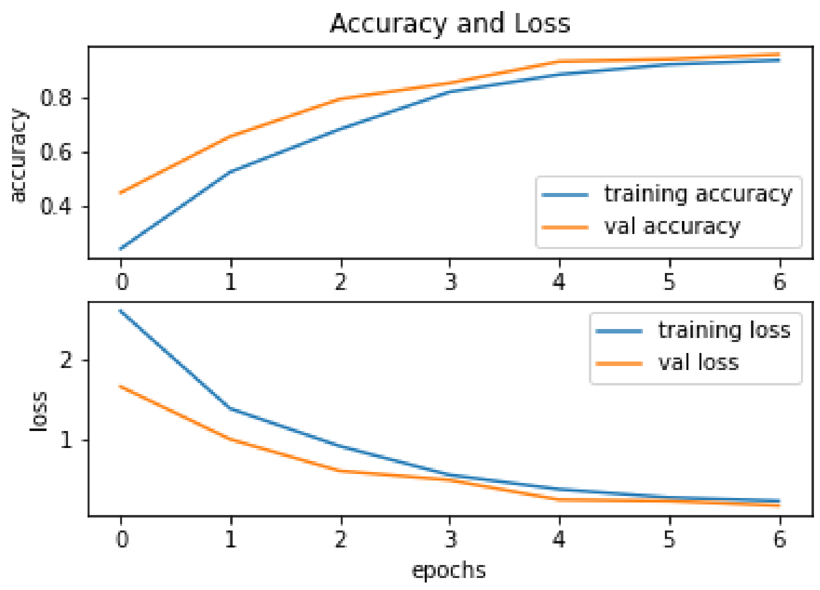


Figure 3. Accuracy and loss of convolutional+recurrent neural network

Here, we got accuracy of 88.6%.

## 4. Experiments with neural networks

Mostly, we will concentrate on CNNs and provide most experiments for them since they are most used and suitable for spatial data. In a nutshell, CNNs are the best option for image and video processing.

### 4.1. ImageNet Large Scale Visual Recognition Challenge models

ImageNet is a large dataset of annotated photographs intended for computer vision research.

The goal of developing the dataset was to provide a resource to promote the research and development of improved methods for computer vision.

The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) is an annual competition in which challenge tasks use subsets of the ImageNetDatset.

The goal of the challenge was to both promote the development of better computer vision techniques and to benchmark the state of the art.

The annual challenge focuses on multiple tasks for "image classification" that includes both assigning a class label to an image based on the main object in the photograph and "object detection" that involves localizing objects within the photograph.

Researchers working on ILSVRC tasks have pushed back the frontier of computer vision research and the methods and papers that describe them are milestones in the fields of computer vision, deep learning, and more broadly in artificial intelligence.

The pace of improvement in the first five years of the ILSVRC was dramatic, perhaps even shocking to the field of computer vision. Success has primarily been achieved by large (deep) convolutional neural networks (CNNs) on graphical processing unit (GPU) hardware, which sparked an interest in deep learning that extended beyond the field out into the mainstream [3].

To sum up, this challenge is very important for image classification tasks. So, we decided to provide some experiments with popular models of this challenge.

We will consider AlexNet, VGG, and ResNet models.

#### 4.1.1. AlexNet

AlexNet CNN is probably one of the simplest methods to approach understanding deep learning concepts and techniques. It is simple enough for beginners and intermediate deep learning practitioners to pick up some good practices of model implementation techniques.

It contains Convolutional, Batch Normalization, MaxPooling, Flatten, and Dense layers. Other used operations and techniques are activation function (ReLU and softmax) and Dropout [4].

In general, all steps are similar to the part 3.1 related to the baseline solution for CNN. That's why later we will describe only the structure of the NN and obtained results.

Initialization:

```
epochs = 20
batch_size = 64
opt = tf.optimizers.SGD(learning_rate=0.001)
loss_func = 'categorical_crossentropy'
metrics=['accuracy']
```

Structure:

```
model.add(Conv2D(filters=96, kernel_size=(3,3), padding='same',
                 activation='relu', input_shape=X_train.shape[1:]))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Conv2D(filters=256, kernel_size=(3,3), padding='same', act
ivation='relu'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Conv2D(filters=384, kernel_size=(3,3), padding='same', act
ivation='relu'))
model.add(BatchNormalization())
model.add(Conv2D(filters=384, kernel_size=(3,3), padding='same', act
ivation='relu'))
model.add(BatchNormalization())
model.add(Conv2D(filters=256, kernel_size=(3,3), padding='same', act
ivation='relu'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(4096, activation='relu'))
model.add(Dropout(rate=0.5))
model.add(Dense(4096, activation='relu'))
model.add(Dropout(rate=0.5))
model.add(Dense(43, activation='softmax'))
```

Results:

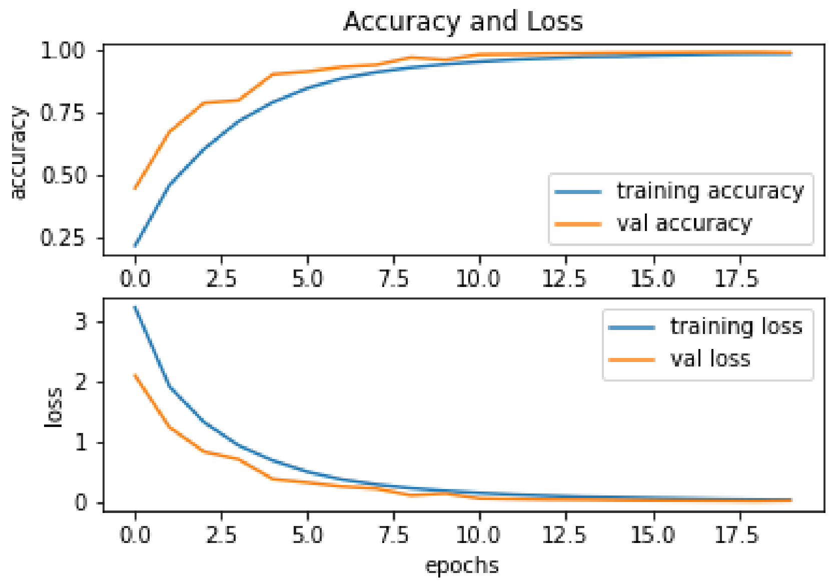


Figure 4. Accuracy and loss of AlexNet

We got accuracy of 92.5%.

### 4.1.2. VGG

VGG is considered to be one of the excellent vision model architectures. The most unique thing about VGG16 is that instead of having a large number of hyper-parameters they focused on having convolution layers of 3x3 filter with a stride 1 and always used the same padding and maxpool layer of 2x2 filter of stride 2. It follows this arrangement of convolution and max pool layers consistently throughout the whole architecture. In the end it has 2 FC (fully connected layers) followed by a softmax for output. The 16 in VGG16 refers to it having 16 layers that have weights. This network is a pretty large network and it has about 138 million (approx) parameters.

Initialization:

```
epochs = 4
batch_size = 32
learning_rate = 1e-3
opt = tf.optimizers.SGD(learning_rate=learning_rate, momentum=0.9, d
ecay=0)
loss_func = 'categorical_crossentropy'
metrics=['accuracy']
```

Structure:

```
model = Sequential()
model.add(Conv2D(filters=64, kernel_size=(3,3), padding='same',
                 activation='relu', input_shape=X_train.shape[1:]))
model.add(Conv2D(filters=64, kernel_size=(3,3), padding='same', acti
vation='relu'))
model.add(MaxPool2D(pool_size=(2,2),padding='same'))
model.add(Conv2D(filters=128, kernel_size=(3,3), padding='same', act
ivation='relu'))
model.add(Conv2D(filters=128, kernel_size=(3,3), padding='same', act
ivation='relu'))
model.add(MaxPool2D(pool_size=(2,2),padding='same'))
model.add(Conv2D(filters=256, kernel_size=(3,3), padding='same', act
ivation='relu'))
model.add(Conv2D(filters=256, kernel_size=(3,3), padding='same', act
ivation='relu'))
model.add(Conv2D(filters=256, kernel_size=(3,3), padding='same', act
ivation='relu'))
model.add(MaxPool2D(pool_size=(2,2),padding='same'))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding='same', act
ivation='relu'))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding='same', act
ivation='relu'))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding='same', act
ivation='relu'))
model.add(MaxPool2D(pool_size=(2,2),padding='same'))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding='same', act
ivation='relu'))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding='same', act
ivation='relu'))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding='same', act
ivation='relu'))
model.add(MaxPool2D(pool_size=(2,2),padding='same'))
model.add(Flatten())
model.add(Dense(4096, activation='relu'))
model.add(Dense(4096, activation='relu'))
model.add(Dense(43, activation='softmax'))
```

Results:

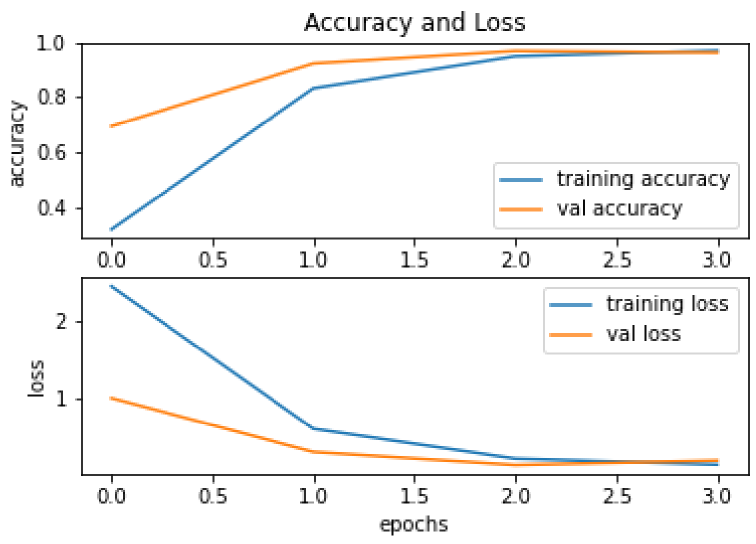


Figure 5. Accuracy and loss of VGG-16

We got accuracy of 92.1%.

### 4.1.3. ResNet-50

In this case, we will try to use NN from the keras applications.

```
from tensorflow.keras.applications import resnet50
```

Initialization:

```
epochs = 10
batch_size = 128
learning_rate = 1e-2
opt = tf.optimizers.SGD(learning_rate=learning_rate, momentum=0.9, d
ecay=0)
loss_func = 'categorical_crossentropy'
metrics=['accuracy']
```

Upload the model:

```
input_image = Input(shape=X_train.shape[1:])
model = tf.keras.applications.ResNet50(include_top=True,
                                       input_tensor=input_image,
                                       weights=None,
                                       input_shape=None,
                                       pooling=None,
                                       classes=43)
```

Results:

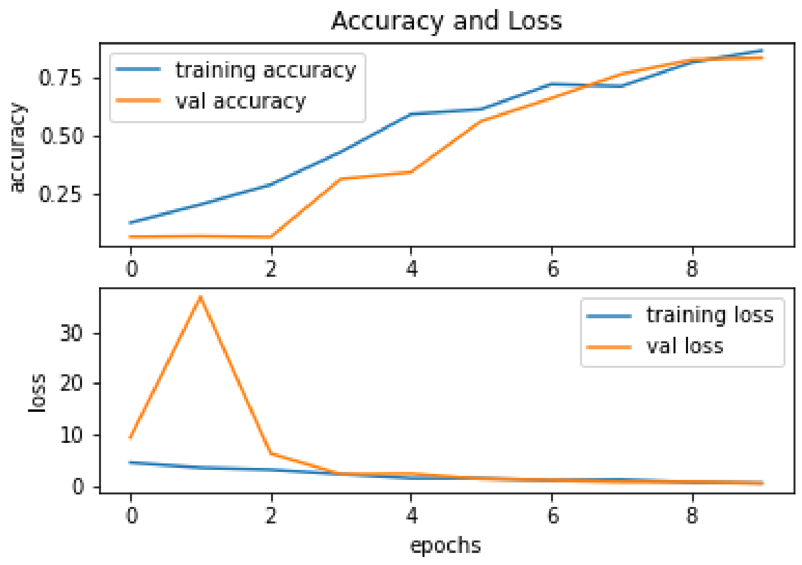


Figure 6. Accuracy and loss of VGG-16

We got an accuracy of 70%. It is the worst model, but with fine tuning this score can be improved dramatically. However, right now it is not the purpose of this work, and we will concentrate on the different topics.

## 4.2. Changing different NN parameters

For this part of the work, we will take a baseline solution for CNN and will change some parameters to achieve better accuracy on the test dataset.

### 4.2.1. Batch size

We decided to look at the dependency of Accuracy and time on batch size.

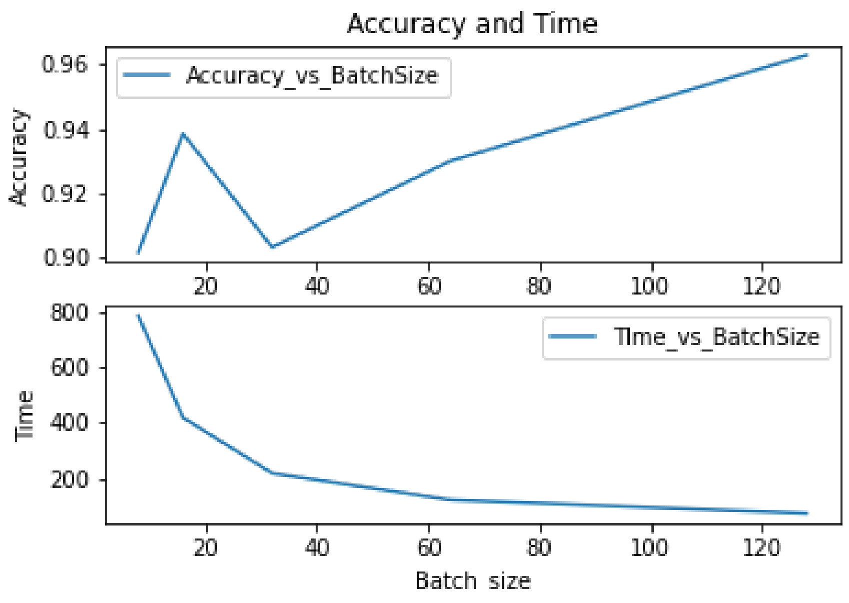


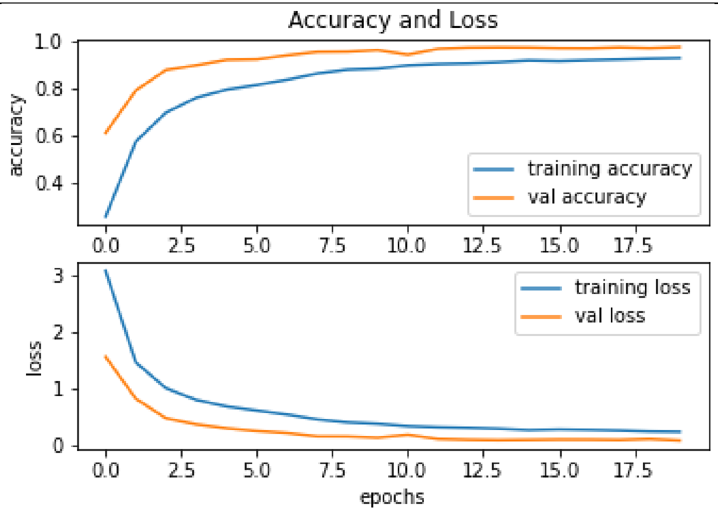
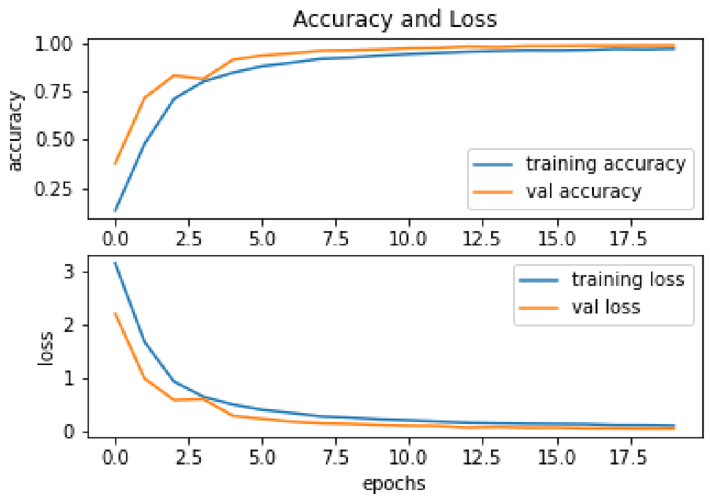
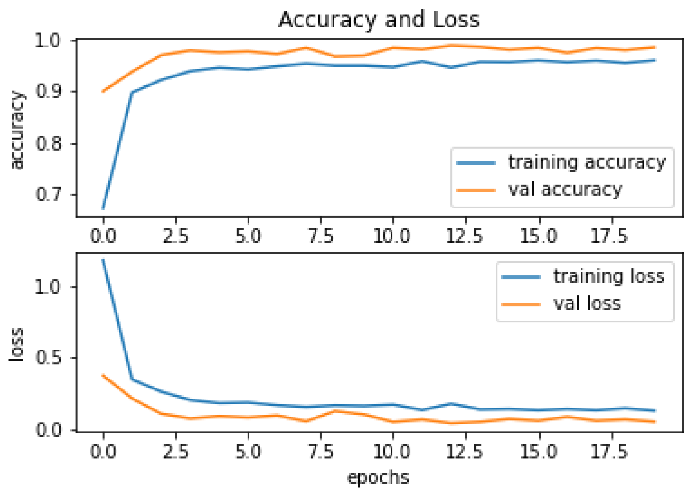
Figure 7. Accuracy and time vs batch size

Usually, the influence of batch size on accuracy depends on the dataset. It is a good idea to consider batch size as a hyperparameter as well when training to find what works best for you and your data. It affects the rate of learning much more. From our plots, we can conclude, that accuracy increases with bigger batch size, and time decreases drastically. These results about accuracy are valid only for our model and dataset. Results related to time are valid in general.

#### 4.2.2. Activation function

For different activation functions (ReLU, Sigmoid, Tanh) let’s check the general performance.

Table 1. Accuracy for different activation functions

| Activation function | Accuracy | Plots |
|---|---|---|
| ReLU | 93.2% |  |
| Sigmoid | 94.3% |  |
| Tanh | 95.2% |  |

So, we can see a slight difference in accuracy between activation functions. Also, activation functions influence the speed of training. However, tanh and sigmoid cause vanishing gradient problem.

#### 4.2.3. Number of epochs

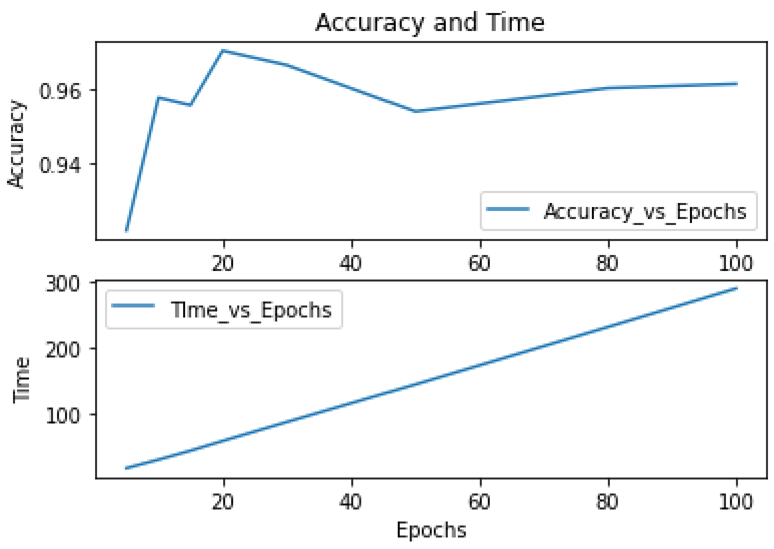


Figure 8.Dependency of Accuracy and time on the number of epochs

As we can see, the maximum is reached with the number of epochs equaled to 20, but after that, it is not changing a lot. With too big number of epochs there is a possibility of overfitting, with small number (less than 20), there is a problem of underfitting.

### 4.3. Check effect of batch normalization and dropout

#### 4.3.1. Batch normalization

Batch normalization is a layer that allows every layer of the network to do learning more independently. It is used to normalize the output of the previous layers. The activations scale the input layer in normalization. Using batch normalization learning becomes efficient also it can be used as regularization to avoid overfitting of the model. The layer is added to the sequential model to standardize the input or the outputs. It can be used at several points in between the layers of the model. It is often placed just after defining the sequential model and after the convolution and pooling layers [5].

We have already used batch normalization when implementing AlexNet CNN. Now, let’s check its effect on the simplified version of the baseline solution to reduce its accuracy.

Simplified structure:

```
model = Sequential()
model.add(Conv2D(filters=32, kernel_size=(5,5), activation='relu', i
nput_shape=X_train.shape[1:]))
model.add(Conv2D(filters=32, kernel_size=(5,5), activation='relu'))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(rate=0.5))
model.add(Dense(43, activation='softmax'))
```

With this structure we got accuracy of 91%.

Structure with BatchNormalization():

```
model = Sequential()
model.add(Conv2D(filters=32, kernel_size=(5,5), activation='relu', i
nput_shape=X_train.shape[1:]))
model.add(BatchNormalization())
model.add(Conv2D(filters=32, kernel_size=(5,5), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2,2)))
model.add(BatchNormalization())
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(rate=0.5))
model.add(Dense(43, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', met
rics=['accuracy'])
```

With BatchNormalization the accuracy is increased and equals 96.2%.

So, we made our model simpler and got the same result as without simplification. This technique improved the accuracy of our model.

Using BatchNormalization with our original model we can still improve the performance and get 98% of accuracy!

Structure:

```
model = Sequential()
model.add(Conv2D(filters=32, kernel_size=(5,5), activation='relu', i
nput_shape=X_train.shape[1:]))
model.add(BatchNormalization())
model.add(Conv2D(filters=32, kernel_size=(5,5), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2,2)))
model.add(BatchNormalization())
model.add(Dropout(rate=0.25))
model.add(Conv2D(filters=32, kernel_size=(3,3), activation='relu'))
model.add(BatchNormalization())
model.add(Conv2D(filters=32, kernel_size=(3,3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2,2)))
model.add(BatchNormalization())
model.add(Dropout(rate=0.25))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(rate=0.5))
model.add(Dense(43, activation='softmax'))
```

#### 4.3.2. Dropout

Dropouts are the regularization technique that is used to prevent overfitting in the model. Dropouts are added to randomly switching some percentage of neurons of the network. When the neurons are switched off the incoming and outgoing connection to those neurons is also switched off. This is done to enhance the learning of the model. Dropouts are usually advised not to use after the convolution layers, they are mostly used after the dense layers of the network. It is always good to only switch off the neurons to 50%. If we switched off more than 50% then there can be chances when the model learning would be poor, and the predictions will not be good [5].

In our case, the change of dropout rate almost does not influence the overall performance. It is because we have a relatively simple model that does not suffer from overfitting.

### 4.4. Transfer learning with fine-tuning approach

Transfer learning is an approach where we use one model trained on a machine learning task and reuse it as a starting point for a different job. Multiple deep learning domains use this approach, including Image Classification, Natural Language Processing, and even Gaming. The ability to adapt a trained model to another task is incredibly valuable.

Once the pre-trained layers have been imported, excluding the “top” of the model, we can take 1 or 2 Transfer learning approaches (Feature Extraction Approach and Fine-Tuning Approach) [6]. Here, we will work with fine-tuning.

We used the ResNet50 model pre-trained on the ImageNet-1000 dataset.

```
conv_base = ResNet50(include_top=False,
                     weights='imagenet',
                     input_shape=input_shape)
```

In order to perform fine-tuning we freeze all layers except last two.

```
for layer in conv_base.layers[:-fine_tune]:
        layer.trainable = False
```

Then, set our own trainable layers and compile the model.

```
top_model = conv_base.output
top_model = Flatten(name="flatten")(top_model)
top_model = Dense(4096, activation='relu')(top_model)
top_model = Dense(1072, activation='relu')(top_model)
top_model = Dropout(0.2)(top_model)
output_layer = Dense(n_classes, activation='softmax')(top_model)
model = Model(inputs=conv_base.input, outputs=output_layer)
model.compile(optimizer=optimizer,
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

We did not have enough time to train this model well to get high accuracy, but the general idea is given. All scripts will be available in the folder with this report, datasets, and NumPy arrays.

## 5. Visualizations

### 5.1. Classes distribution

First, let's look at the distribution of classes.

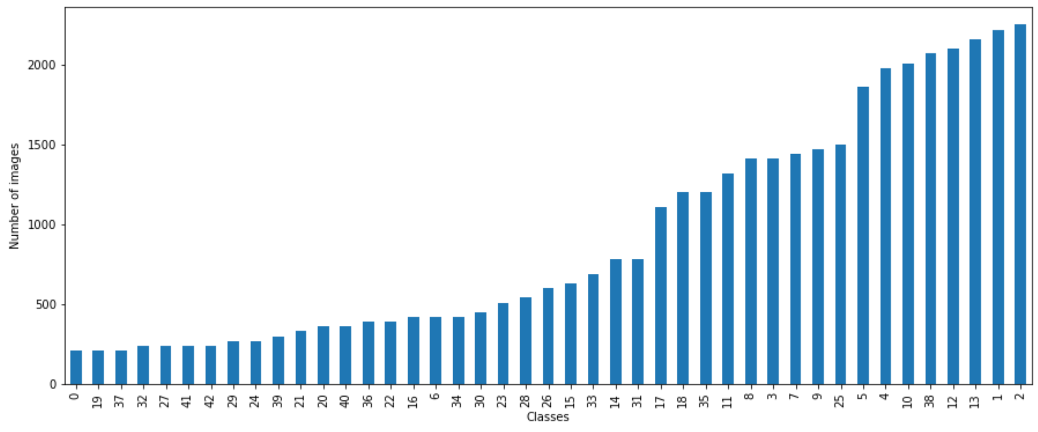
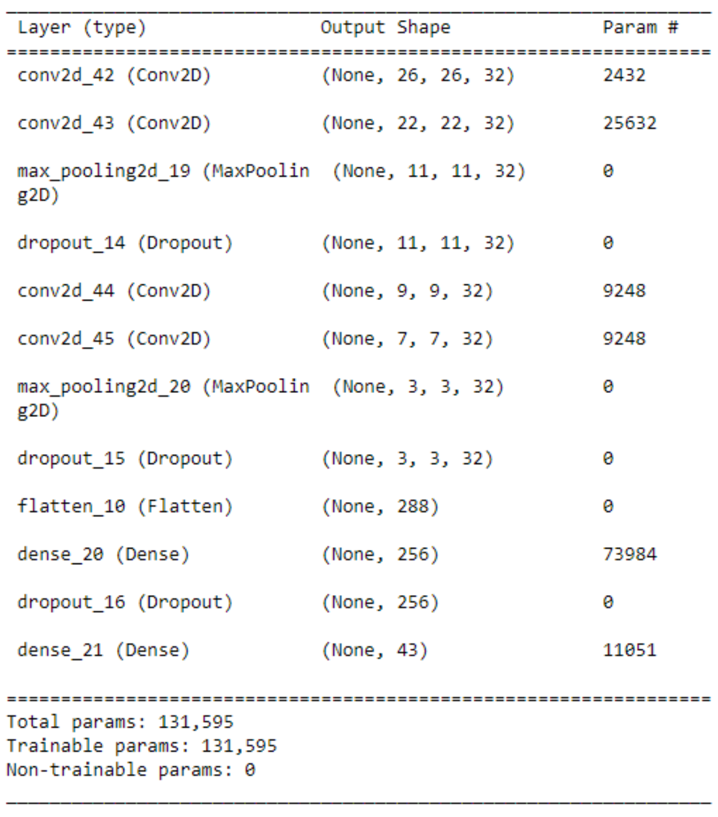


Figure 9. Classes distribution

As we can see, the dataset is unbalanced.

### 5.2. Model summary

We can print a table with all layers and parameters of the model.



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_42 (Conv2D)          (None, 26, 26, 32)        2432

 conv2d_43 (Conv2D)          (None, 22, 22, 32)        25632

 max_pooling2d_19 (MaxPoolin  (None, 11, 11, 32)       0
 g2D)

 dropout_14 (Dropout)        (None, 11, 11, 32)        0

 conv2d_44 (Conv2D)          (None, 9, 9, 32)          9248

 conv2d_45 (Conv2D)          (None, 7, 7, 32)          9248

 max_pooling2d_20 (MaxPoolin  (None, 3, 3, 32)         0
 g2D)

 dropout_15 (Dropout)        (None, 3, 3, 32)          0

 flatten_10 (Flatten)        (None, 288)               0

 dense_20 (Dense)            (None, 256)               73984

 dropout_16 (Dropout)        (None, 256)               0

 dense_21 (Dense)            (None, 43)                11051

=================================================================
Total params: 131,595
Trainable params: 131,595
Non-trainable params: 0
_________________________________________________________________
```

Figure 10. Model summary

## 6. Information about other activities

There are some things that we planned to do but due to lack of time did not.

- We tried to visualize convolutional layers using articles [7-9] but did not manage to do that
- We wanted to do more experiments with hyperparameters
- Planned adding data augmentation to improve the unbalanced dataset
- We wanted to do some experiments with transfer learning using different pre-trained models and their parameters. Also, as for transfer learning, it would be great to try another approach – feature extraction
- Try to do some experiments with feed forward NN and recurrent NN even though they are less used in Image Classification projects

## 7. Learning outcomes and results

This project was extremely useful for my understanding of many neural network concepts, not only basic but also advanced. Previously, I worked mostly with classical machine learning algorithms and projects (mostly data given in tables). Here, I tried another big part of artificial intelligence. It was not only helpful but very exciting as well. I learned a lot by reading class presentations, different blogs dedicated to neural networks, scientific articles and so on.

The most interesting part was related to transfer learning. The concept on his own is interesting and useful for different applications. Also, I liked recurrent neural networks and after this project, I am planning to do some projects related to natural language processing and time series.

From the model performance perspective, the most useful feature was batch normalization. It increased the accuracy a lot (up to 98%).

During the project I also noticed, the importance of GPU and optimization of the prepared model. It can take so much time if the model is not optimized, the learning rate, number of epochs and batch size were chosen incorrectly, and GPU is not available. Without GPU it is useless to compute anything. Probably, it works only for very simple models and for complex models GPU is a vital tool.

The most useful part is the main part about convolutional neural networks and all experiments related to them. Since it is the state-of-art technique for image and video processing and in particular image classification, I am quite sure that I will use some of these concepts in my future projects. After this semester I decided to go much deeper into machine learning and artificial intelligence and even find a job as an ML/AI engineer.

References:

[1] “Autonomous Traffic Sign (ATSR) Detection and Recognition using Deep CNN”, Danyah A. Alghmghama, Ghazanfar Latif, Jaafar Alghazo, Loay Alzubaidi, 2019

[2] https://www.kaggle.com/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign

[3] https://machinelearningmastery.com/introduction-to-the-imagenet-large-scale-visual-recognition-challenge-ilsvrc/

[4] https://towardsdatascience.com/implementing-alexnet-cnn-architecture-using-tensorflow-2-0-and-keras-2113e090ad98

[5] https://analyticsindiamag.com/everything-you-should-know-about-dropouts-and-batchnormalization-in-cnn/

[6] https://www.learndatasci.com/tutorials/hands-on-transfer-learning-keras/

[7] https://www.analyticsvidhya.com/blog/2018/03/essentials-of-deep-learning-visualizing-convolutional-neural-networks/

[8] https://poloclub.github.io/cnn-explainer/

[9] https://towardsdatascience.com/how-to-visualize-convolutional-features-in-40-lines-of-code-70b7d87b0030

[10] https://machinelearningmastery.com/transfer-learning-for-deep-learning/

[11] https://data-flair.training/blogs/python-project-traffic-signs-recognition/