Deep Learning – Project I
Convolutional Neural Networks

Michał Korwek, Emil Łasocha

April 1, 2025

Contents

1 Methodology and preliminaries

The dataset used in the project was a CINIC10 dataset, which consist of 270 000 images. It was split into equal subsets: train, validation and test. Each of them contain 90 000 images.

The tests were conducted partly on NVIDIA GeForce GTX 1650 and partly on Google Colab's GPU T4. The CNNs were trained using PyTorch library.

In order to reproduce results, for files `Project1 fewshot new.ipynb` and `Dropout Softvoting.ipynb`:
1. Download the dataset as .zip file from
https://www.kaggle.com/datasets/mengcius/cinic10
2. Upload it to Google Colab
3. Make a new code-block with *!unzip archive.zip* command.

In order to reproduce results, for files `DL Project1.ipynb`, `DataAugmentation.ipynb`, `LearningRate.ipynb`, `plots.ipynb` and `Simple CNN.ipynb` you must put the files and dataset in one folder (current path to dataset used in every of these files is cinic10/versions/1/)

2 Influence of hyper-parameter change related to training process.

2.1 Learning rate

The first hyperparameter we were evaluating is learning rate. This number decides, what will be the pace of learning, deciding about the weight and power of backpropagation process. In this experiment, we wanted to evaluate, what is the optimal rate for efficientnet b0 model. We use ready-made model from torchvision models package, without pre-training, but we update the last layer, to have 10 outputs, as number of labels in our data. We repeat the experiment 3 times on different random seeds (0,1,2) to achieve statistical significance. Each processes training lasted 20 epochs. Batch size was equal to 128. The results looks as follows:

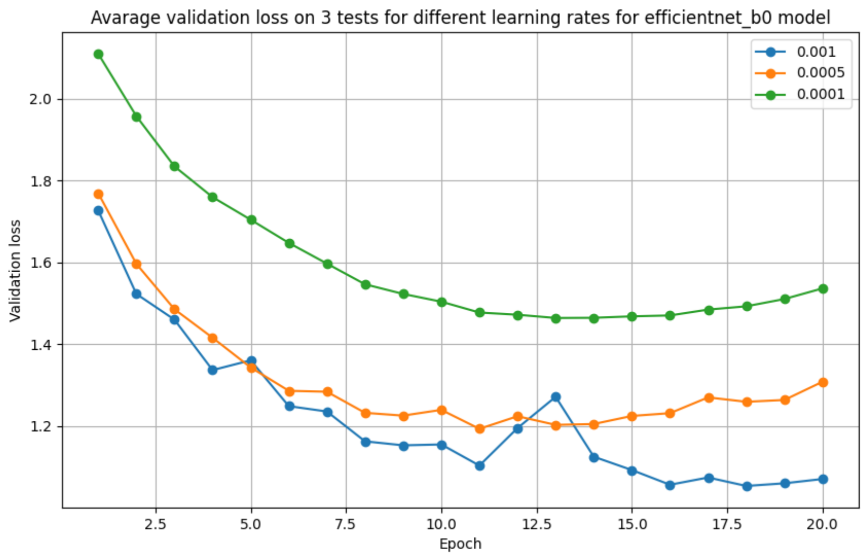


Figure 1: Learning rate, average validation loss

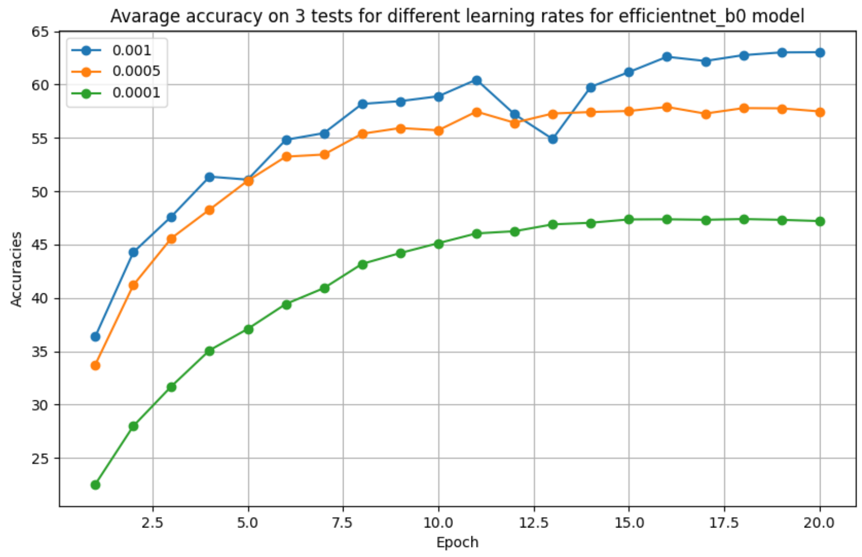


Figure 2: Learning rate, average validation accuracy

We can see, that the best results are for learning rate equal to 0.001,

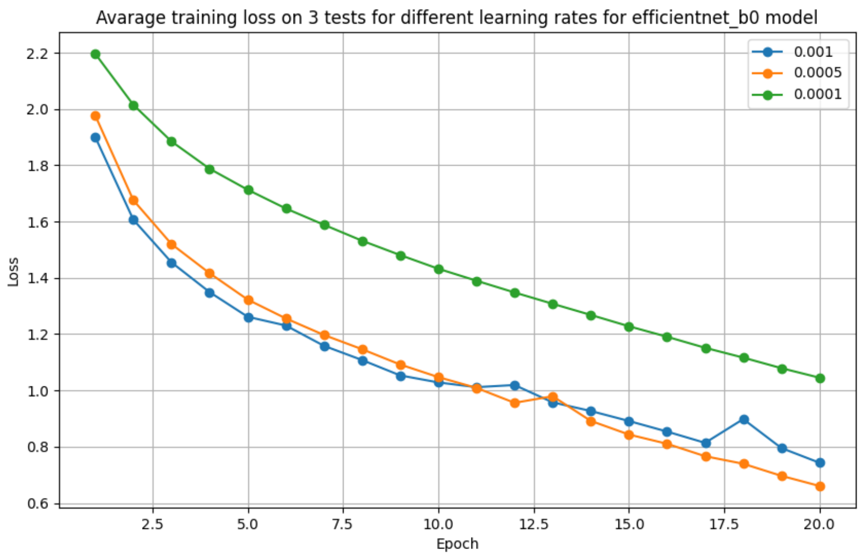


Figure 3: Learning rate, average train loss

with around 64 % accuracy on validation set. We can see, that for all values, train losses were decreasing in stable way, but validation losses get stable or even started to increse what can mean we get to the point where further continuation of the experiment would possible cause overfitting.

We have also tried testing more learning rates, but without repeating experiment on different architecture - ResNet. We have used ready-implemented model from pytorch models resnext50 32x4d. As the model is prepared for pictures in 256x256 resolution, we have applied slight modifications in first and last layer. Our changes:

```
model.fc = nn.Linear(2048, num_labels, bias = True)
model.conv1 = nn.Conv2d(3,64, kernel_size = (3,3), stride = 1,
padding = 1, bias = False)
```

Original first layer:

```
(conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2),
padding=(3, 3), bias=False)
```

Max number of epochs was set to 10 however neither of models acheived it due to the stop rule. We have set that the stop rule will be 2 times in row increase in validation loss. Results are as follows:

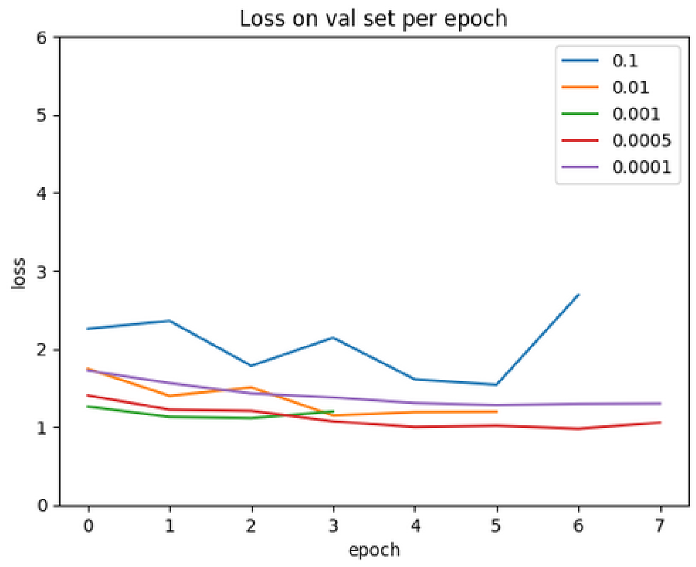


Figure 4: ResNet validation loss

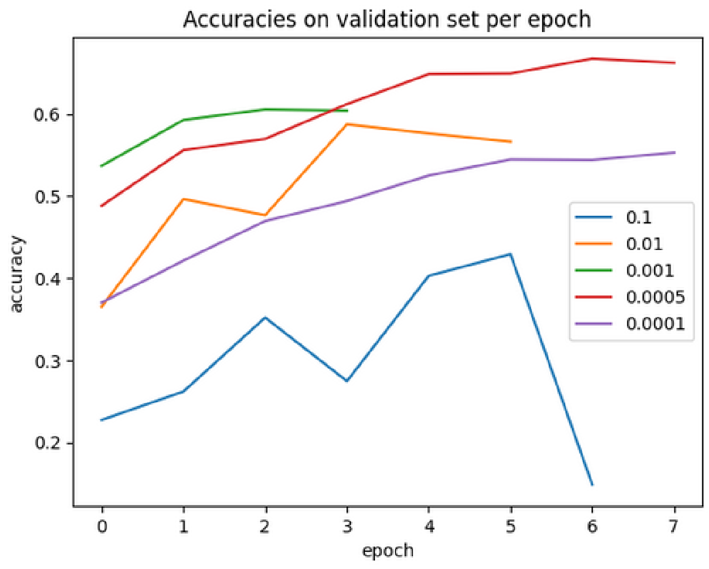


Figure 5: ResNet validation accuracies

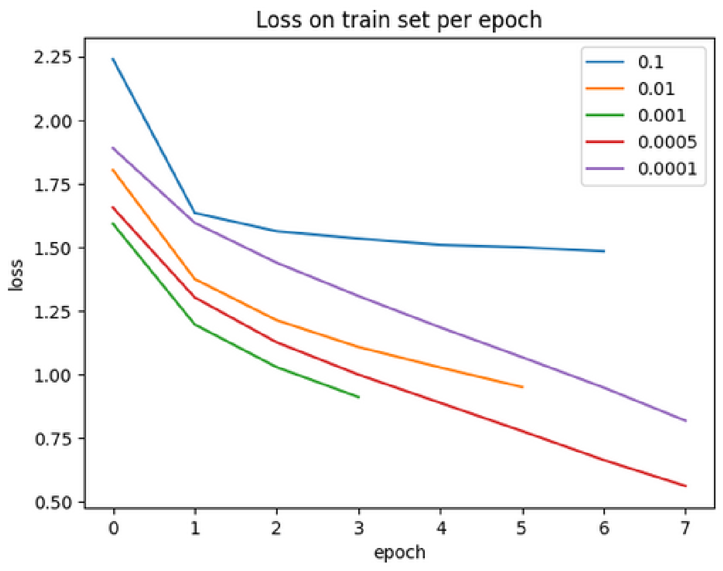
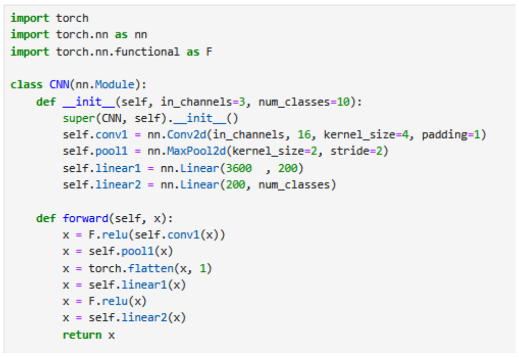


Figure 6: ResNet train loss

Results are slightly different than previously. We get to the conclusion, that learning rate is not an universal value, and its optimal value can differ between models. What is more, setting such restrict stopping rule, quickly finishes training process but we have no doubt, the results may be more interesting and valuable if the train process lasted longer. However, some models achieved similiar accuracy than previously tested EfficientNet even faster. Unfortunately, we are lest with two hypothesis, either this architecture is way better than efficientNet and it is possible to get way better results with it and the second, that it is a shallow model and we simply achieved its limits fast. The models look as follows:

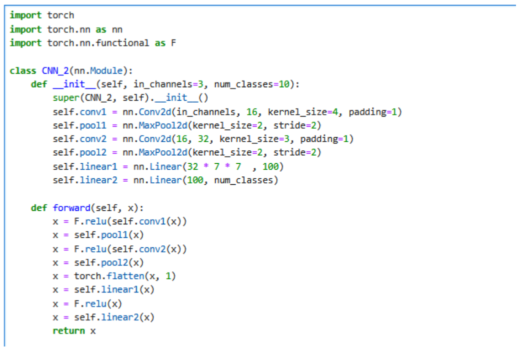


```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class CNN(nn.Module):
    def __init__(self, in_channels=3, num_classes=10):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, 16, kernel_size=4, padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.linear1 = nn.Linear(3600  , 200)
        self.linear2 = nn.Linear(200, num_classes)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)
        x = torch.flatten(x, 1)
        x = self.linear1(x)
        x = F.relu(x)
        x = self.linear2(x)
        return x
```

Figure 7: CNN

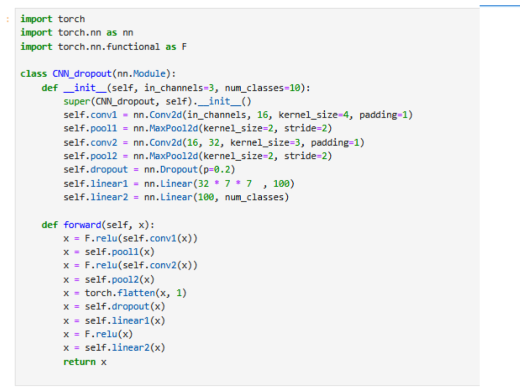


```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class CNN_2(nn.Module):
    def __init__(self, in_channels=3, num_classes=10):
        super(CNN_2, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, 16, kernel_size=4, padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.linear1 = nn.Linear(32 * 7 * 7  , 100)
        self.linear2 = nn.Linear(100, num_classes)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)
        x = F.relu(self.conv2(x))
        x = self.pool2(x)
        x = torch.flatten(x, 1)
        x = self.linear1(x)
        x = F.relu(x)
        x = self.linear2(x)
        return x
```

Figure 8: CNN_2

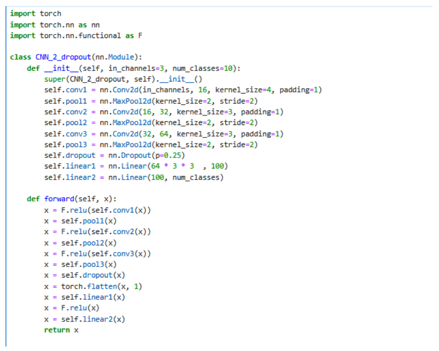


```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class CNN_dropout(nn.Module):
    def __init__(self, in_channels=3, num_classes=10):
        super(CNN_dropout, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, 16, kernel_size=4, padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.dropout = nn.Dropout(p=0.2)
        self.linear1 = nn.Linear(32 * 7 * 7  , 100)
        self.linear2 = nn.Linear(100, num_classes)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)
        x = F.relu(self.conv2(x))
        x = self.pool2(x)
        x = torch.flatten(x, 1)
        x = self.dropout(x)
        x = self.linear1(x)
        x = F.relu(x)
        x = self.linear2(x)
        return x
```

Figure 9: CNN_dropout



```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class CNN_2_dropout(nn.Module):
    def __init__(self, in_channels=3, num_classes=10):
        super(CNN_2_dropout, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, 16, kernel_size=4, padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.dropout = nn.Dropout(p=0.25)
        self.linear1 = nn.Linear(64 * 3 * 3  , 100)
        self.linear2 = nn.Linear(100, num_classes)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)
        x = F.relu(self.conv2(x))
        x = self.pool2(x)
        x = F.relu(self.conv3(x))
        x = self.pool3(x)
        x = self.dropout(x)
        x = torch.flatten(x, 1)
        x = self.linear1(x)
        x = F.relu(x)
        x = self.linear2(x)
        return x
```

Figure 10: CNN_2_dropout

The training time was set to 60 epochs, with stop condition being 4 times in row increase in val loss. The results look as follows

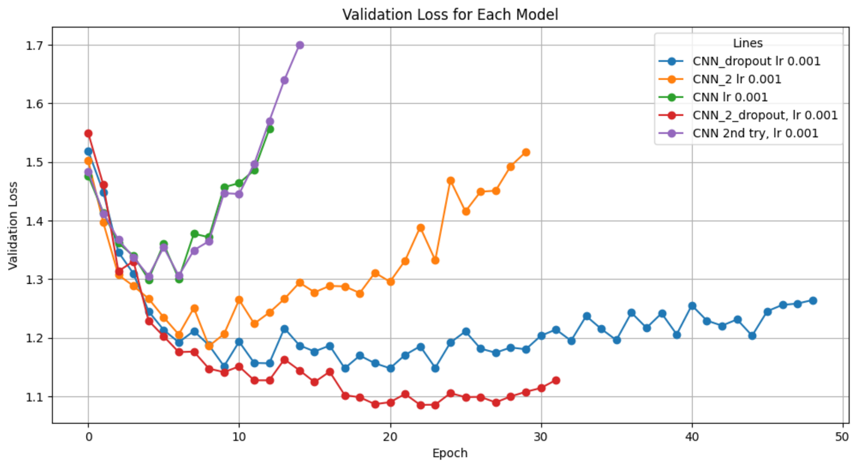


Figure 11: Simple CNNs, validation loss

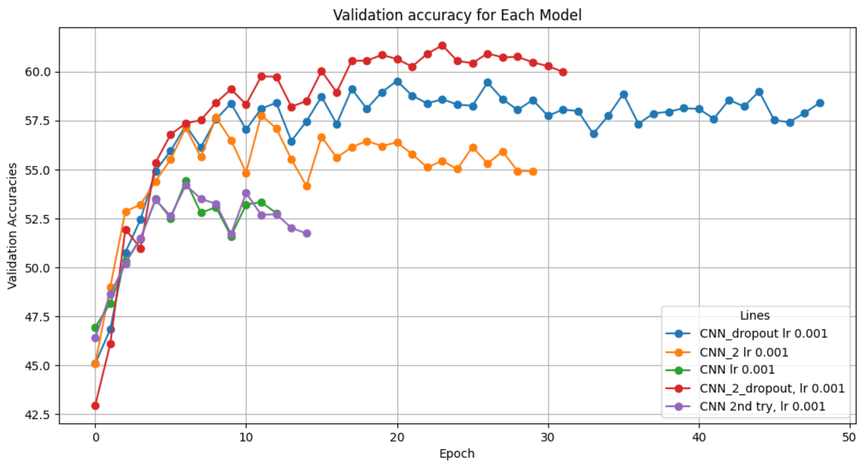


Figure 12: Simple CNNs, validation accuracy

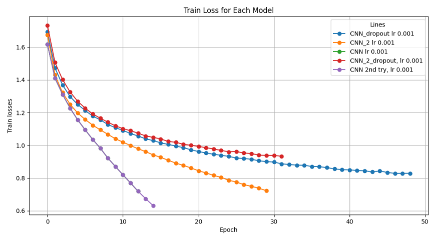


Figure 13: Simple CNNs, train loss

It seems that neural nets with dropout achieved better results than the same models without this technique. We can assume that it prevents overfitting well and provides good generalisation. We will get back to these models when we will consider advanced data augmentation techniques.

3 Dropout hyper-parameter testing with soft-voting ensemble

The next experiment, which can be found in `Dropout Softvoting.ipynb` file, was conducted on custom Convolutional Neural Network model. Dropout is not a global estimation parameter and good practices uses bigger dropouts at later layers, mostly after ReLU layer. We also take into consideration the fact, that the values of Dropout should belong to the interval $[0.2, 0.5]$. Smaller dropout values might not prevent overfitting of the model, while on the other hand – bigger dropout values might result in undertfitting. The hyperparameters taken into account are:

- Model 1 – dropout values: $\{0.2, 0.3, 0.4, 0.5\}$
- Model 2 – dropout values: $\{0.2, 0.25, 0.3, 0.35\}$
- Model 3 – dropout values: $\{0.35, 0.4, 0.45, 0.5\}$

Additionally, below is provided the full models architecture:

```
model1 = nn.Sequential(
    # Block 1: 32x32x3 -> 16x16x32
    nn.Conv2d(3, 32, krnl=3, pad=1),
    nn.BatchNorm2d(32),
    nn.ReLU(),
    nn.Conv2d(32, 32, krnl=3, pad=1),
    nn.BatchNorm2d(32),
    nn.ReLU(),
    nn.MaxPool2d(2),
    nn.Dropout(0.2),

    # Block 2: 16x16x32 -> 8x8x64
    nn.Conv2d(32, 64, krn=3, pad=1),
    nn.BatchNorm2d(64),
    nn.ReLU(),
    nn.Conv2d(64, 64, krn=3, pad=1),
    nn.BatchNorm2d(64),
    nn.ReLU(),
    nn.MaxPool2d(2),
    nn.Dropout(0.3),

    # Block 3: 8x8x64 -> 4x4x128
    nn.Conv2d(64, 128, krn=3, pad=1),
    nn.BatchNorm2d(128),
    nn.ReLU(),
    nn.Conv2d(128, 128, krn=3, pad=1),
    nn.BatchNorm2d(128),
    nn.ReLU(),
    nn.MaxPool2d(2),
    nn.Dropout(0.4),

    # Classifier
    nn.Flatten(),
    nn.Linear(128 * 4 * 4, 512),
    nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(512, 10)
)
```

```
model2 = nn.Sequential(
# Block 1: 32x32x3 -> 16x16x32
nn.Conv2d(3, 32, krn=3, pad=1),
nn.BatchNorm2d(32),
nn.ReLU(),
nn.Conv2d(32, 32, krn=3, pad=1),
nn.BatchNorm2d(32),
nn.ReLU(),
nn.MaxPool2d(2),
nn.Dropout(0.2),

# Block 2: 16x16x32 -> 8x8x64
nn.Conv2d(32, 64, krn=3, pad=1),
nn.BatchNorm2d(64),
nn.ReLU(),
nn.Conv2d(64, 64, krn=3, pad=1),
nn.BatchNorm2d(64),
nn.ReLU(),
nn.MaxPool2d(2),
nn.Dropout(0.25),

# Block 3: 8x8x64 -> 4x4x128
nn.Conv2d(64, 128, krn=3, pad=1),
nn.BatchNorm2d(128),
nn.ReLU(),
nn.Conv2d(128, 128, krn=3, pad=1),
nn.BatchNorm2d(128),
nn.ReLU(),
nn.MaxPool2d(2),
nn.Dropout(0.3),

# Classifier
nn.Flatten(),
nn.Linear(128 * 4 * 4, 512),
nn.ReLU(),
nn.Dropout(0.35),
nn.Linear(512, 10)
)
```

```
model3 = nn.Sequential(
    # Block 1: 32x32x3 -> 16x16x32
    nn.Conv2d(3, 32, krn=3, pad=1),
    nn.BatchNorm2d(32),
    nn.ReLU(),
    nn.Conv2d(32, 32, krn=3, pad=1),
    nn.BatchNorm2d(32),
    nn.ReLU(),
    nn.MaxPool2d(2),
    nn.Dropout(0.35),

    # Block 2: 16x16x32 -> 8x8x64
    nn.Conv2d(32, 64, krn=3, pad=1),
    nn.BatchNorm2d(64),
    nn.ReLU(),
    nn.Conv2d(64, 64, krn=3, pad=1),
    nn.BatchNorm2d(64),
    nn.ReLU(),
    nn.MaxPool2d(2),
    nn.Dropout(0.4),

    # Block 3: 8x8x64 -> 4x4x128
    nn.Conv2d(64, 128, krn=3, pad=1),
    nn.BatchNorm2d(128),
    nn.ReLU(),
    nn.Conv2d(128, 128, krn=3, pad=1),
    nn.BatchNorm2d(128),
    nn.ReLU(),
    nn.MaxPool2d(2),
    nn.Dropout(0.45),

    # Classifier
    nn.Flatten(),
    nn.Linear(128 * 4 * 4, 512),
    nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(512, 10)
)
```

Batch Size was set to 64, which balances GPU memory constraints. Validation/test data remains unshuffled for consistent evaluation. The model was trained using the Adam optimizer with a learning rate of 0.003. We employed the Cross-Entropy Loss, which is standard for multi-class classification tasks. Each model was trained on 50 epochs. The training of each model on Google Colab's GPU for approximately $50 \cdot 2\text{mins} = 100\text{mins}$.

Model 1

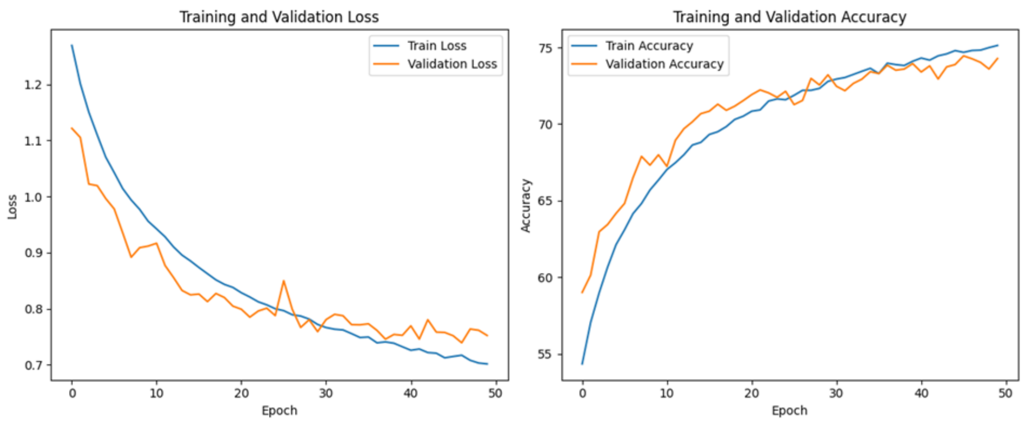
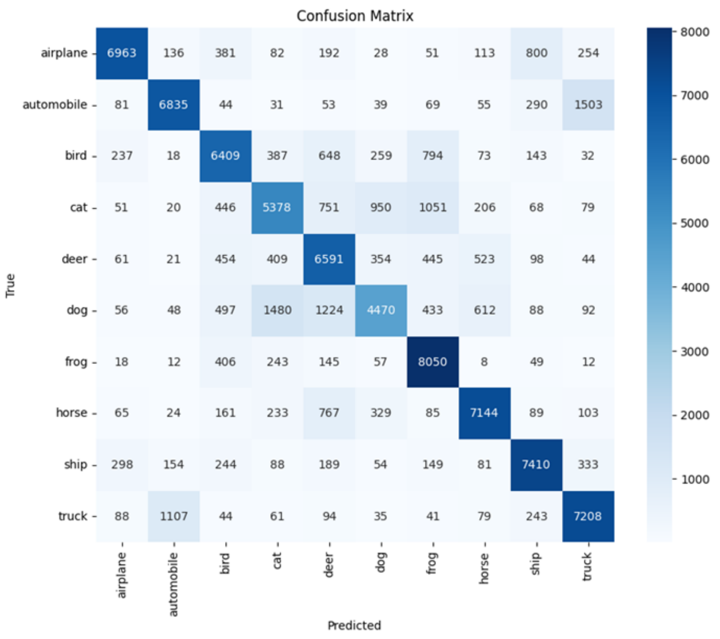


Figure 14: Metrics of Model 1



True \ Predicted	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	6963	136	381	82	192	28	51	113	800	254
automobile	81	6835	44	31	53	39	69	55	290	1503
bird	237	18	6409	387	648	259	794	73	143	32
cat	51	20	446	5378	751	950	1051	206	68	79
deer	61	21	454	409	6591	354	445	523	98	44
dog	56	48	497	1480	1224	4470	433	612	88	92
frog	18	12	406	243	145	57	8050	8	49	12
horse	65	24	161	233	767	329	85	7144	89	103
ship	298	154	244	88	189	54	149	81	7410	333
truck	88	1107	44	61	94	35	41	79	243	7208

Figure 15: Confusion Matrix of Model 1

Final Test Results: Loss: 0.7603 | Accuracy: 73.84%

We can see that the train loss is less than the validation loss at the end of the training. Both training and validation accuracy are high on later epochs. The confusion matrix shows that each class is being recognized and there are no anomalies. The worst performing class is class with image of "dog", as it was sometimes mistaken with deer or cat, which also seems normal. With reasonable test accuracy, we conclude, that the dropout values from the Model 1, i.e. $[0.2, 0.3, 0.4, 0.5]$ are sensible hyperparameters. It means, that checking both lower and higher values of them are good choice of later experiments.

Model 2

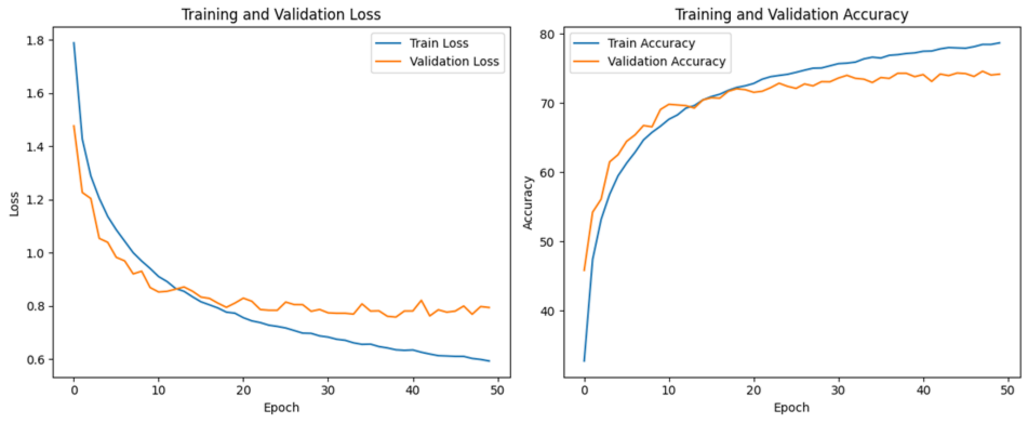


Figure 16: Metrics of Model 2

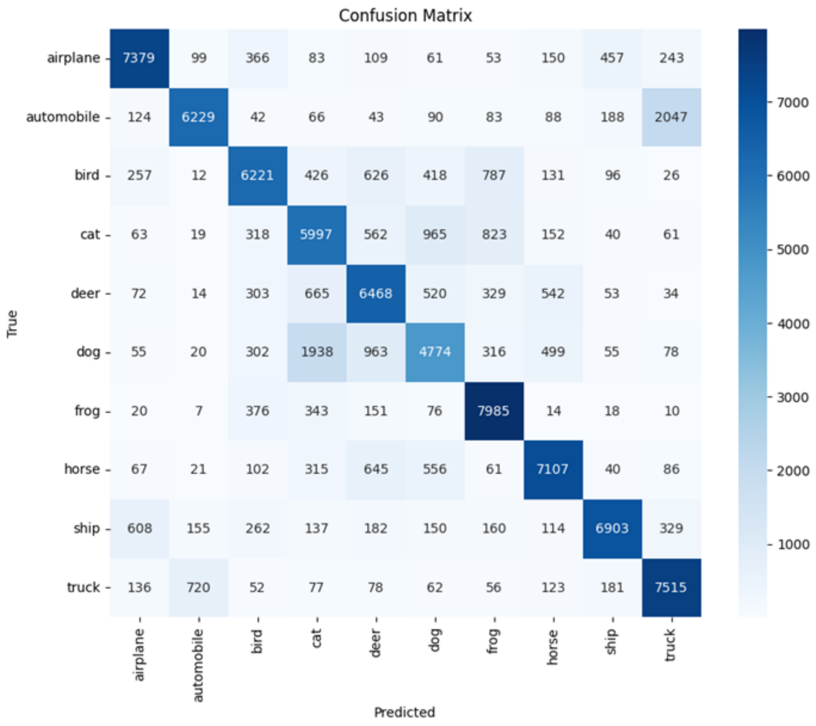


Figure 17: Confusion Matrix of Model 2

Final Test Results: Loss: 0.8057 | Accuracy: 73.98%

This model, on the other hand used dropout hypermateres from the set $[0.2, 0.25, 0.3, 0.35]$. Due to lower values, we can deduce that it could lead to overfitting, which indeed is the case. The training loss is much smaller than the validation loss and the training accuracy is slightly bigger than validation accuracy. Nonetheless, the confusion matrix and the test results show that the model performs neatly.

Model 3

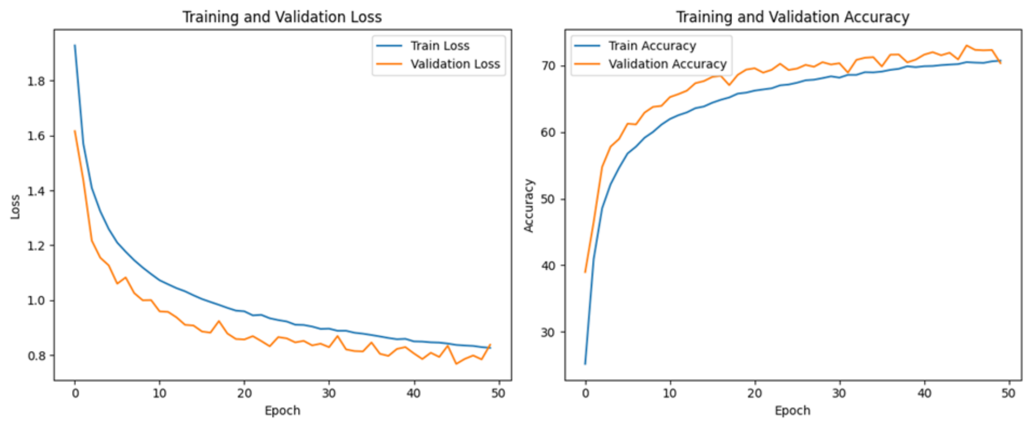
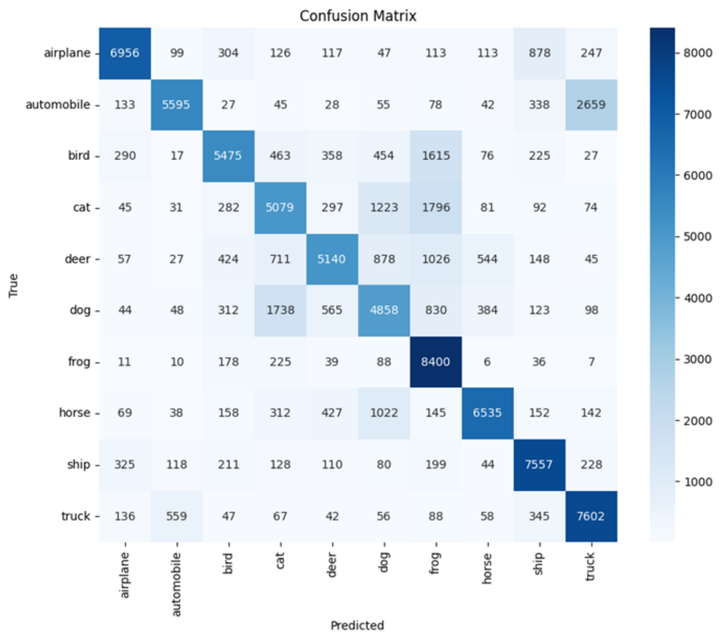


Figure 18: Metrics of Model 3



True \ Predicted	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	6956	99	304	126	117	47	113	113	878	247
automobile	133	5595	27	45	28	55	78	42	338	2659
bird	290	17	5475	463	358	454	1615	76	225	27
cat	45	31	282	5079	297	1223	1796	81	92	74
deer	57	27	424	711	5140	878	1026	544	148	45
dog	44	48	312	1738	565	4858	830	384	123	98
frog	11	10	178	225	39	88	8400	6	36	7
horse	69	38	158	312	427	1022	145	6535	152	142
ship	325	118	211	128	110	80	199	44	7557	228
truck	136	559	47	67	42	56	88	58	345	7602

Figure 19: Confusion Matrix of Model 3

Final Test Results: Loss: 0.8438 | Accuracy: 70.22%

Now, the hyperparameters range is $[0.35, 0.4, 0.45, 0.5]$, which suggest that model could underfitting. It indeed is the case, showing that the validation loss is smaller than training loss throughout the whole training. Similarly, nearly throughout all the training, the validation accuracy is bigger than training accuracy. Overall, the model performs slightly worse than previous two.

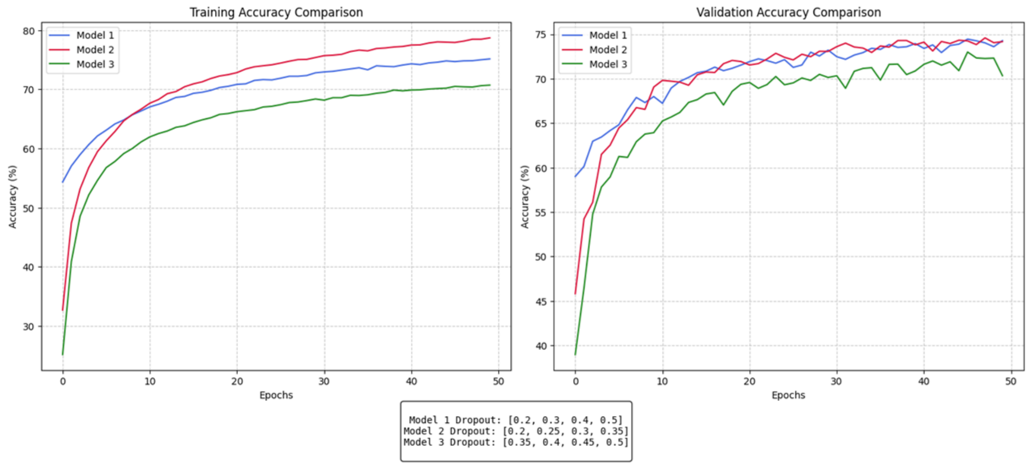


Figure 20: Comparison of all three models

Soft-voting ensemble

Based on the previous three models, we would like to create a soft-voting ensemble. Instead of each model voting for a single class, they output class probabilities, which are averaged. We consider the following three cases regarding the weights we and pick the best one. Due to the third model being slightly worse, we ensure that its weigh is not that significant.

- Case 1 – weights of consecutive models: $[0.45, 0.45, 0.1]$
- Case 2 – weights of consecutive models: $[0.6, 0.3, 0.1]$
- Case 3 – weights of consecutive models: $[0.3, 0.6, 0.1]$

The results on the test data are as follows:

- Case 1 Weighted Ensemble: Loss=1.7660, Accuracy=75.94%

- Case 2 Weighted Ensemble: Loss=1.7686, Accuracy=75.77%
- Case 3 Weighted Ensemble: Loss=1.7638, Accuracy=75.59%

We see that they are all pretty much similar. What is important, that the test accuracy is higher by about 2 percentage point than without the soft-voting ensemble.

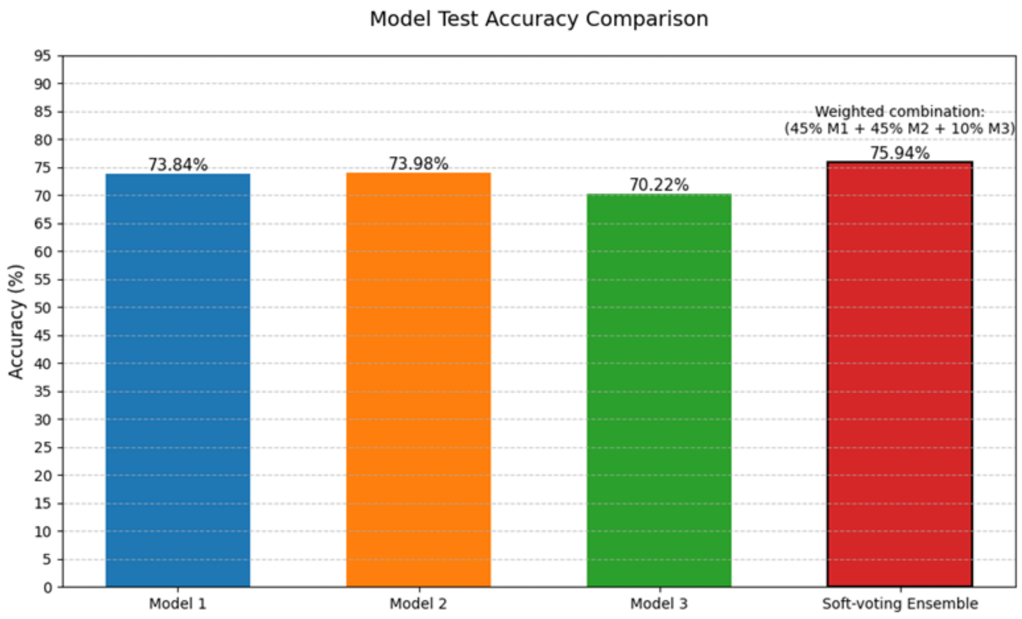


Figure 21: Comparison of performance on test dataset

4 Influence of data augmentions

In this section we will explore different data augmentations. This methods allow us to create new, unseen data what can be valuable in trainning process and generalisation of our model.

4.1 Standard data augmentations

We focused on a few simple data augmentation techniques: rotations, greyscale, scaling and different joints of them. Rotations, as the name suggests, rotates the picture. It was extremely interessing, if the degree of rotation or the colour of gaps fill will make a difference. That's why we have prepared an experiment where we used not-pretrained EfficientNet to check different rotation types. It is the list of transformation we have checked:

```
    transform_rotations = transforms.Compose([
    transforms.RandomRotation(degrees = (-10,10), fill = 255),
    transforms.ToTensor()
])
transform_rotations_black = transforms.Compose([
    transforms.RandomRotation(degrees = (-10,10)),
    transforms.ToTensor()
])

transform_rotations_2 = transforms.Compose([
    transforms.RandomRotation(degrees = (-30,30), fill = 255),
    transforms.ToTensor()
])

transform_rotations_2_black = transforms.Compose([
    transforms.RandomRotation(degrees = (-30,30)),
    transforms.ToTensor()
])
transform_rotations_3 = transforms.Compose([
    transforms.RandomRotation(degrees = (-60,60), fill = 255),
    transforms.ToTensor()
])
transform_rotations_3_black = transforms.Compose([
    transforms.RandomRotation(degrees = (-60,60)),
    transforms.ToTensor()
])
transform_rotations_4 = transforms.Compose([
    transforms.RandomRotation(degrees = (-120,120), fill = 255),
    transforms.ToTensor()
])
transform_rotations_5 = transforms.Compose([
    transforms.RandomRotation(degrees = (-180,180), fill = 255),
    transforms.ToTensor()
])
```

Some of them, are named 'black'. The picture will look like Figure 15,

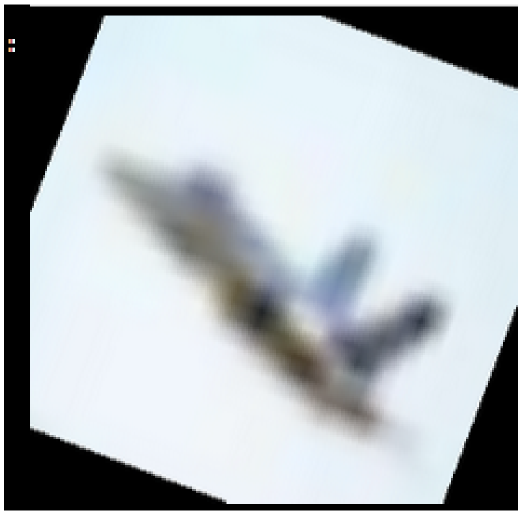


Figure 22: Rotated plane with black backround

pictures without this note, will look like Figure 16

The experiment is conducted once, each model is trained 10 epochs, with stop condition of 2 times in row validation loss increase. The results looks as follows:

As we can see, small rotations provide us with better results. What is more, it seems that the color of the background, as we want to expect, does not change much in training results. Models which saw huge variety of rotations did not get good results, so it seems either pictures rotated this well did not provide well generalization, or the provided dataset is diverse enough to achieve good results.

Another experiment, was comparing all prepared data augmentation methods on pretrained EfficientNet. We have all abovementioned data augmentation methods with mixing between them, the prepared transforms look as follows:

```
    transform_default = transforms.Compose([transforms.ToTensor()])

transform_greyscale = transforms.Compose([
    transforms.Grayscale(num_output_channels=3),
    transforms.ToTensor()
])

transform_rotations = transforms.Compose([
```

Figure 23: Rotated plane with default backround

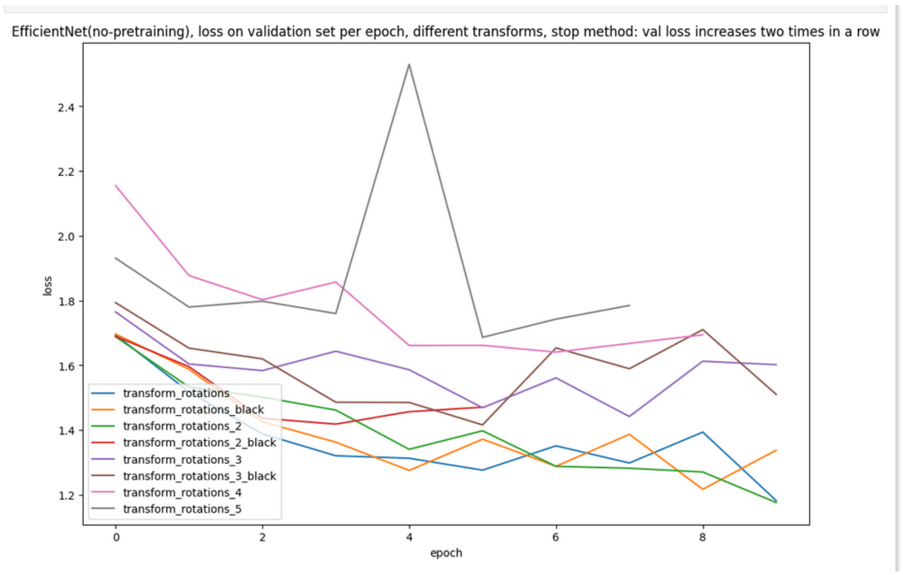


Figure 24: Rotations, losses on validation set

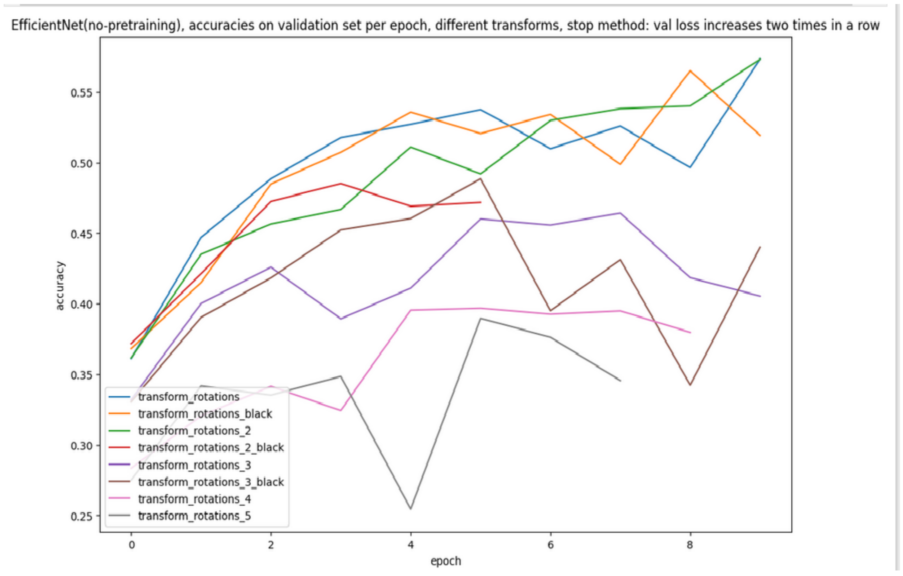


Figure 25: Rotations, accuracies on validation set

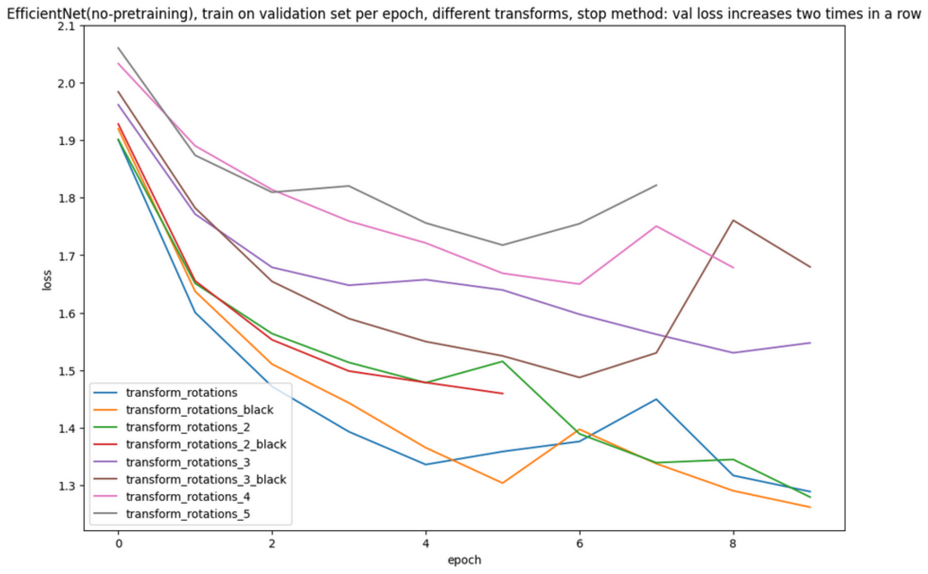


Figure 26: Rotations, losses on train set

```
    transforms.RandomRotation(degrees = (-180,180)),
    transforms.ToTensor()
])
transform_scaling = transforms.Compose([
    transforms.RandomResizedCrop(size = (32,32)),
    transforms.ToTensor()
])

transform_greyscale_rotations = transforms.Compose([
    transforms.RandomRotation(degrees = (-180,180)),
    transforms.Grayscale(num_output_channels=3),
    transforms.ToTensor()
])

transform_greyscale_scaling = transforms.Compose([
    transforms.Grayscale(num_output_channels=3),
    transforms.RandomResizedCrop(size = (32,32)),
    transforms.ToTensor()
])

transform_rotations_scaling = transforms.Compose([
    transforms.RandomRotation(degrees = (-180,180)),
    transforms.RandomResizedCrop(size = (32,32)),
    transforms.ToTensor()
])

transform_all =  transforms.Compose([
    transforms.Grayscale(num_output_channels=3),
    transforms.RandomRotation(degrees = (-180,180)),
    transforms.RandomResizedCrop(size = (32,32)),
    transforms.ToTensor()
])
```

What is important, models prepared on greyscale pictures are validated on pictures in greyscale as well, the other models are validated on default validation set with any additional augmentations. The experiment was repeated once, 15 epochs train each, stop condition: validation loss increase two times in a row.

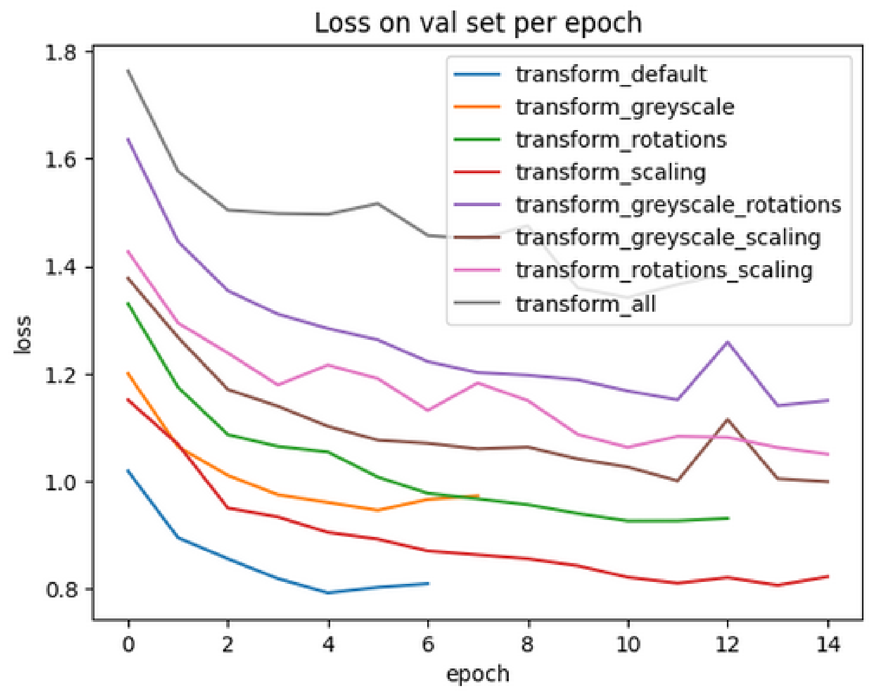


Figure 27: Different data augmentation loss on validation set

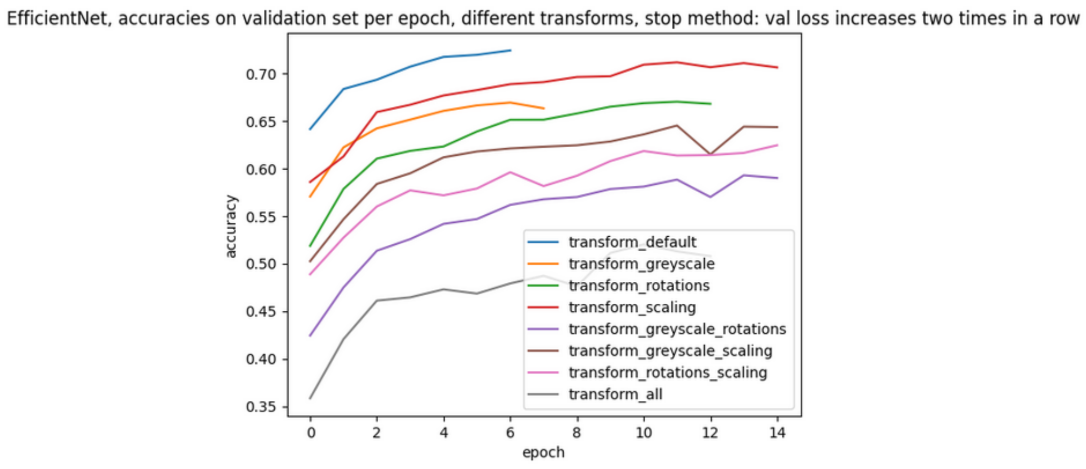


Figure 28: Different data augmentation accuracy on validation set

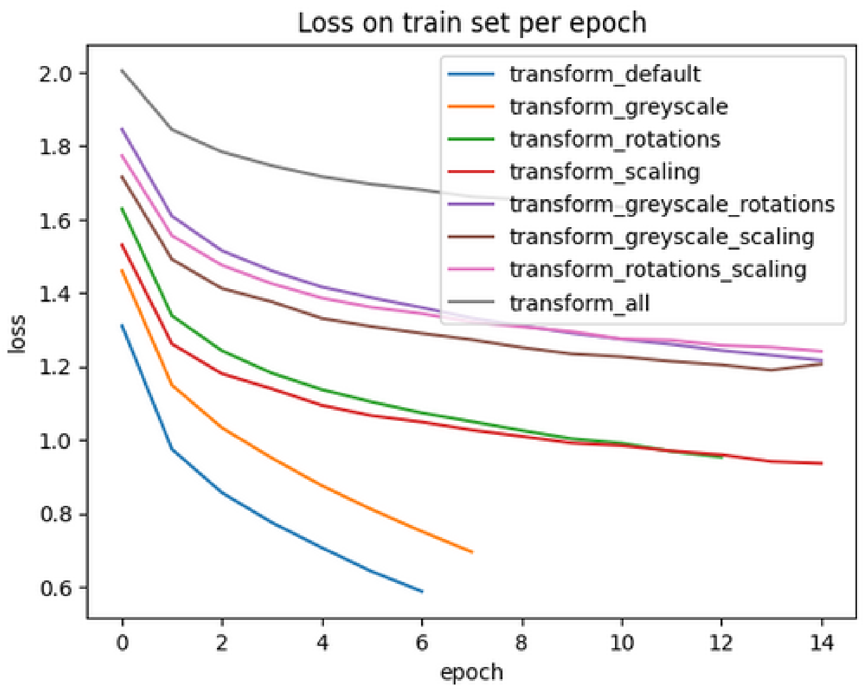


Figure 29: Different data augmentation loss on train set

Surprisingly, the less amount of augmentations, the better results are. Only data scaling can achieve similiar results to default data, while other methods doesn't imply much progress. One of suggestion which arose during evaluating results, was that pre-trained model it too trained too catch such stuff and it only will be fitting to new categories it just saw. Because of that, we decided to try once again, but on not pretrained model.

The experiment was conducted in similiar environment. The only difference is not pretrained EfficientNet model, which will be trained:

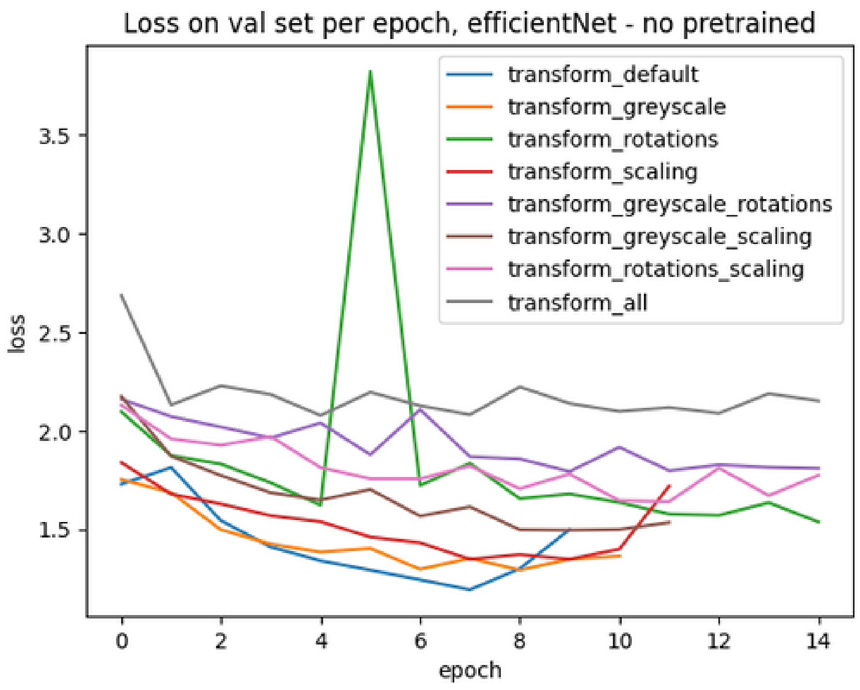


Figure 30: Different data augmentation loss on validation set

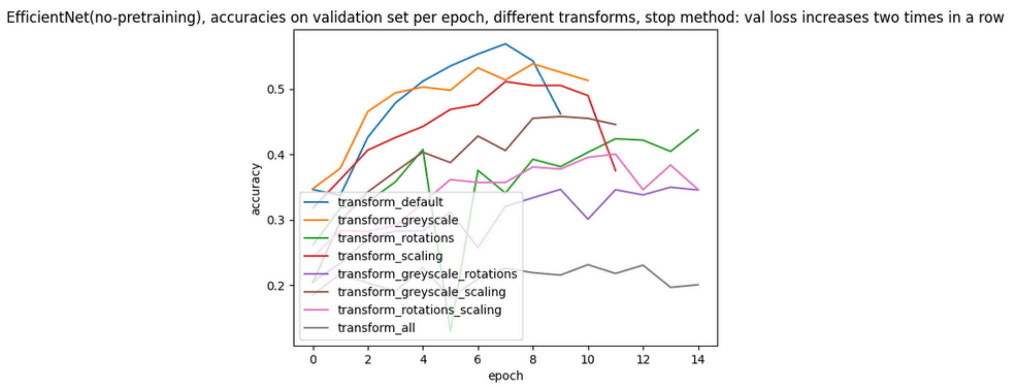


Figure 31: Different data augmentation accuracy on validation set

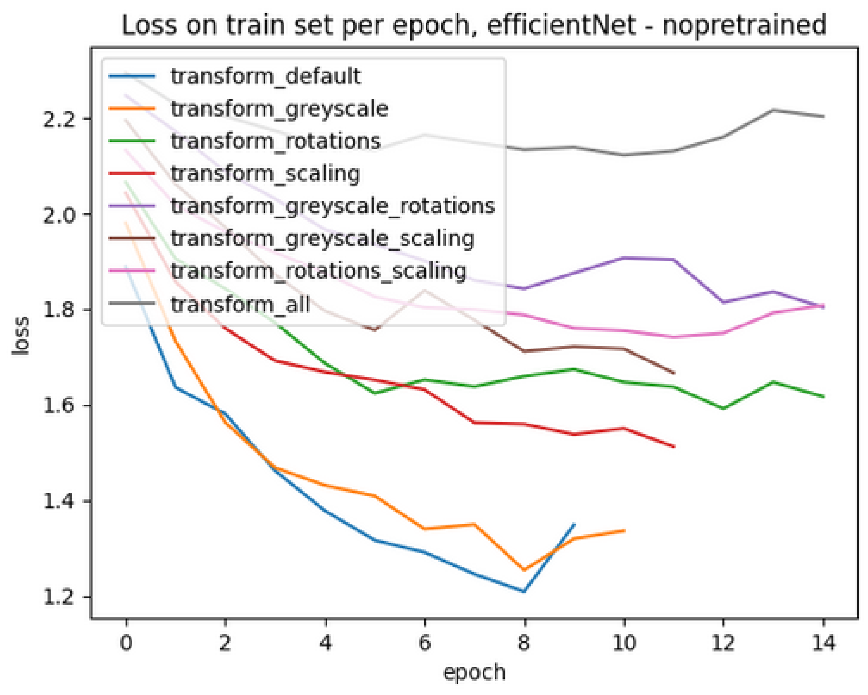


Figure 32: Different data augmentation loss on train set

The results are not satisfactory. Accuracy is not higher than 0.6 in any case. The train loss is rather flat, models seem to almost stay in one place and don't make progress. The conclusion can be made, that efficientNet model, works the best on default data. Another idea arose, maybe if we mix some default pictures with augmentated one, then we will have better results. According to this approach, the next set of models have been trained. Stop condition is 3 times in row val loss increase, number of epochs increase to 20, the rest of the parameters stay the same. Set of transforms used in this approach looks like this:

```
    transform_default = transforms.Compose([transforms.ToTensor()])

transform_greyscale = transforms.Compose([
    transforms.Grayscale(num_output_channels=3),
    transforms.ToTensor()
])
```

```
transform_rotations = transforms.Compose([
    transforms.RandomRotation(degrees = (-180,180)),
    transforms.ToTensor()
])
transform_scaling = transforms.Compose([
    transforms.RandomResizedCrop(size = (32,32), scale=(0.8, 1.0)),
    transforms.ToTensor()
])

transform_greyscale_rotations = transforms.Compose([
    transforms.RandomRotation(degrees = (-180,180)),
    transforms.Grayscale(num_output_channels=3),
    transforms.ToTensor()
])

transform_greyscale_scaling = transforms.Compose([
    transforms.Grayscale(num_output_channels=3),
    transforms.RandomResizedCrop(size = (32,32), scale=(0.8, 1.0)),
    transforms.ToTensor()
])

transform_rotations_scaling = transforms.Compose([
    transforms.RandomRotation(degrees = (-180,180)),
    transforms.RandomResizedCrop(size = (32,32), scale=(0.8, 1.0)),
    transforms.ToTensor()
])

transform_all =  transforms.Compose([
    transforms.Grayscale(num_output_channels=3),
    transforms.RandomRotation(degrees = (-180,180)),
    transforms.RandomResizedCrop(size = (32,32), scale=(0.8, 1.0)),
    transforms.ToTensor()
])


transform_random = transforms.Compose([
    transforms.ToTensor(),
        transforms.RandomApply(torch.nn.ModuleList([
     transforms.RandomRotation(degrees = (-180,180)),
 ]), p=0.2),
    transforms.RandomApply(torch.nn.ModuleList([
     transforms.RandomResizedCrop(size = (32,32), scale=(0.8, 1.0)),
 ]), p=0.2)])


transform_random_2 = transforms.Compose([
    transforms.ToTensor(),
        transforms.RandomApply(torch.nn.ModuleList([
     transforms.RandomRotation(degrees = (-180,180)),
 ]), p=0.4),
    transforms.RandomApply(torch.nn.ModuleList([
     transforms.RandomResizedCrop(size = (32,32), scale=(0.8, 1.0)),
 ]), p=0.4)])

transform_random_3 = transforms.Compose([
    transforms.ToTensor(),
        transforms.RandomApply(torch.nn.ModuleList([
     transforms.RandomRotation(degrees = (-180,180)),
 ]), p=0.75),
    transforms.RandomApply(torch.nn.ModuleList([
     transforms.RandomResizedCrop(size = (32,32), scale=(0.8, 1.0)),
 ]), p=0.75)])
```

In 'random' transforms, some random transforms are used on pictures. Depending on the random number, with 0.2, 0.4 or 0.75 probability the picture is rotated and with the same probability it is randomly scaled. The results look a bit more promising:

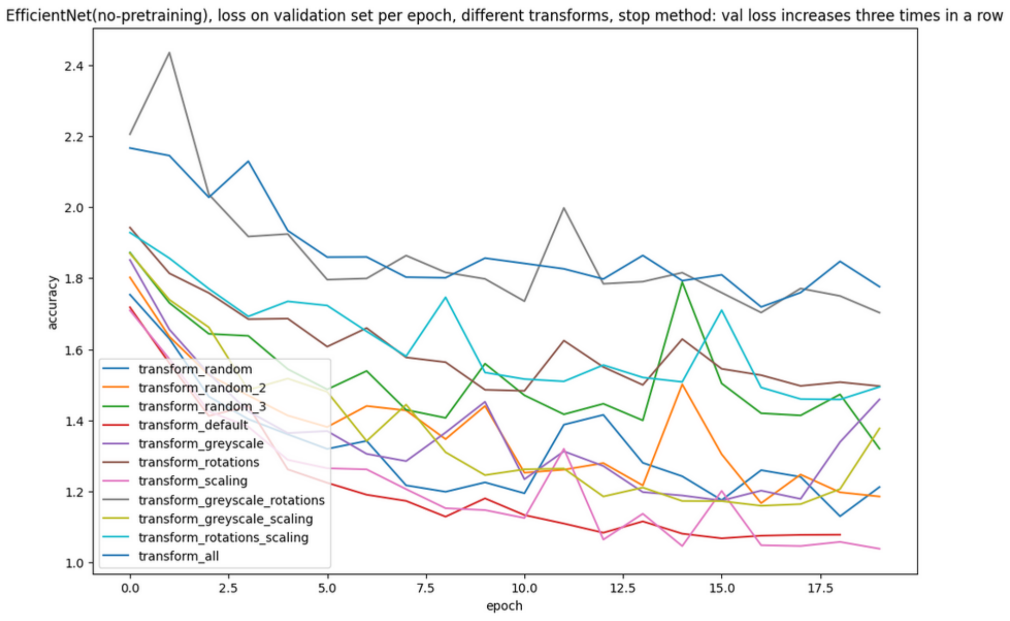


Figure 33: Different data augmentation loss on validation set

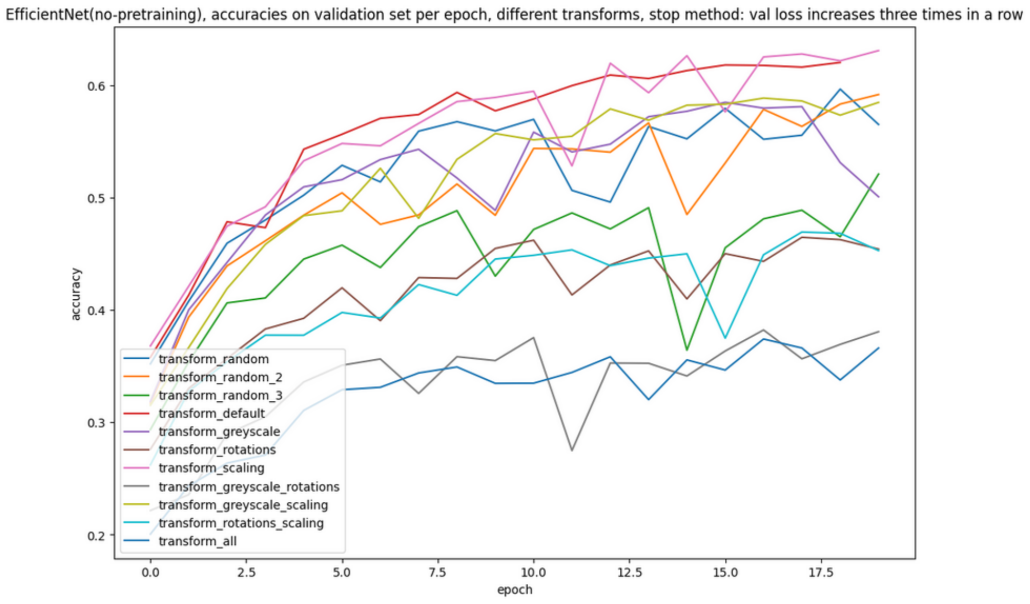


Figure 34: Different data augmentation accuracy on validation set

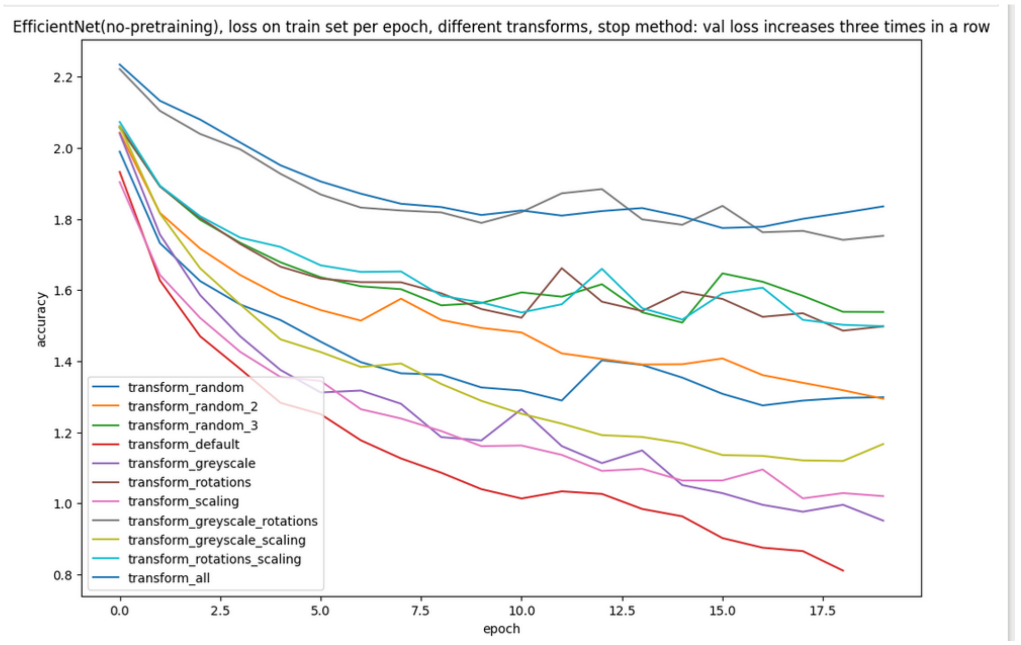


Figure 35: Different data augmentation loss on train set

Default transformation gives good results again. What is interesting, scalling gives the highest accuracy in this example. Our test with random transforms seems to be successful, as it gets high accuracy on validation set but not as high as default transformations. What is worth highlighitng, all greyscale transforms don't perform well. It is the perfect moment to introduce the last augmentation technique we used in the learning process.

4.2 More advanced data augmentations

We decided to use CutMix. The data augmentation technique, which mixes two pictures into one. It allows the model to focus more on different characteristics of classified categories and study faster. We connected testing it, with already working solutions. In the upcoming experiment we will train not pretrained ElasticNet again. Batch size 128, learning rate 0.001, stop condition is 3 times in row validation loss increase, max epochs 12. The experiment was be repeated 3 times on different seed, taking in general almost 20 hours to complete. The results are presented in diagrams:

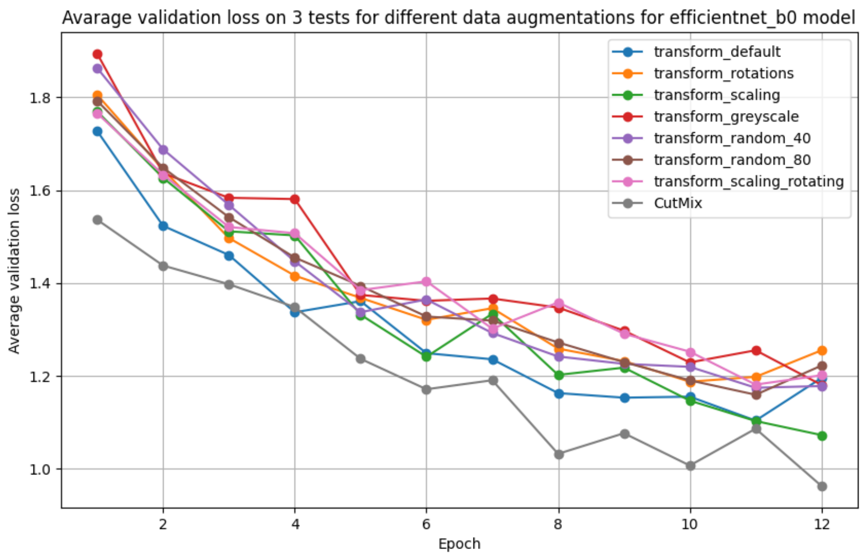


Figure 36: Validation loss - data augmentation with cutMix

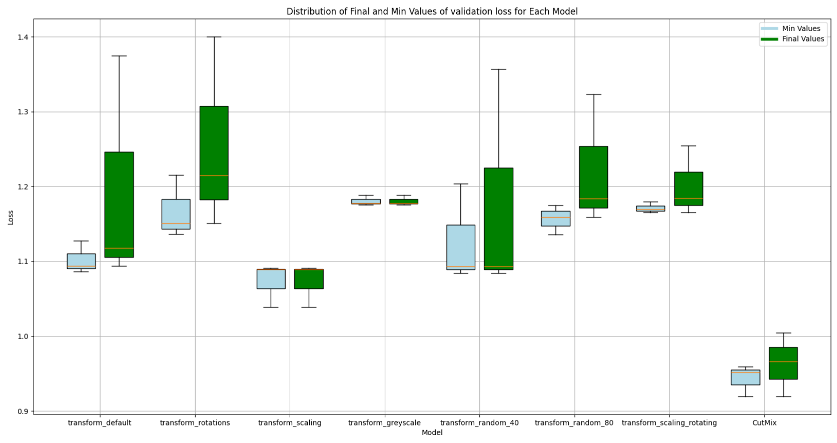


Figure 37: Validation loss - data augmentation with cutMix

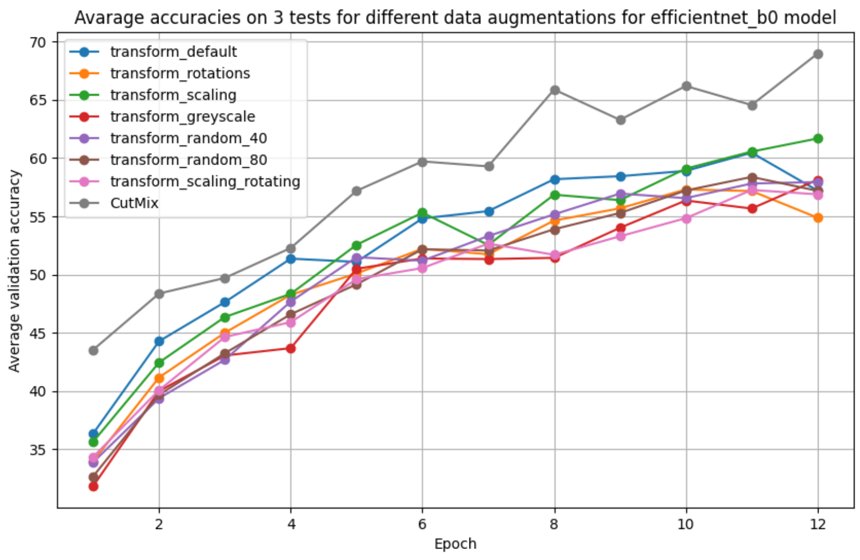


Figure 38: Validation accuracy - data augmentation with cutMix

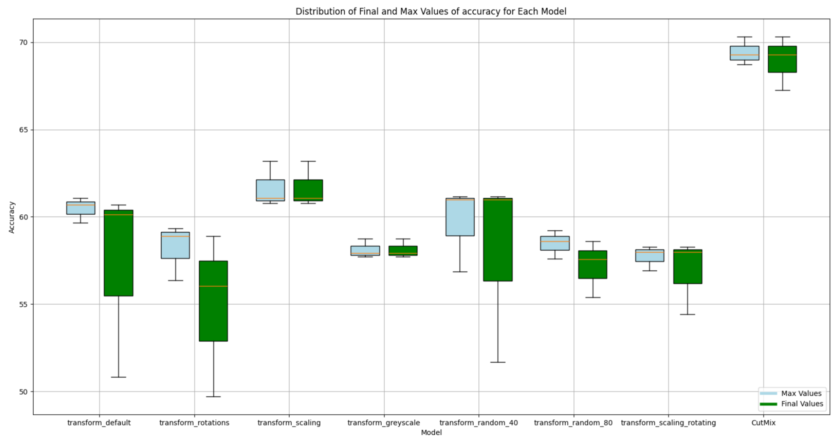


Figure 39: Validation accuracy - data augmentation with cutMix

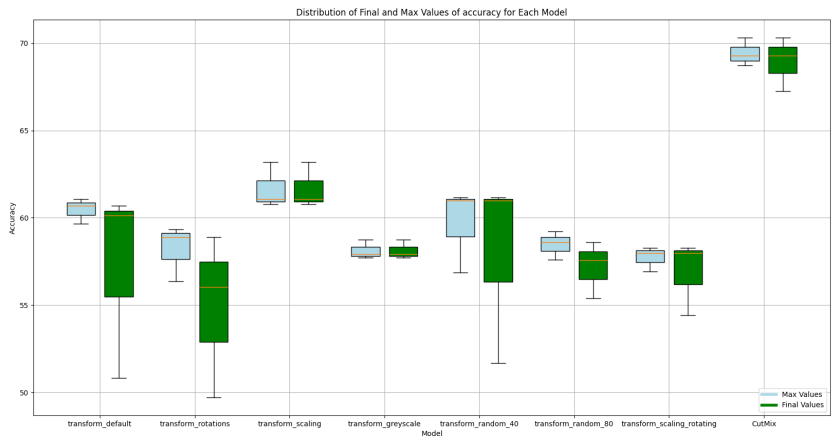


Figure 41: train loss - data augmentation with cutMix

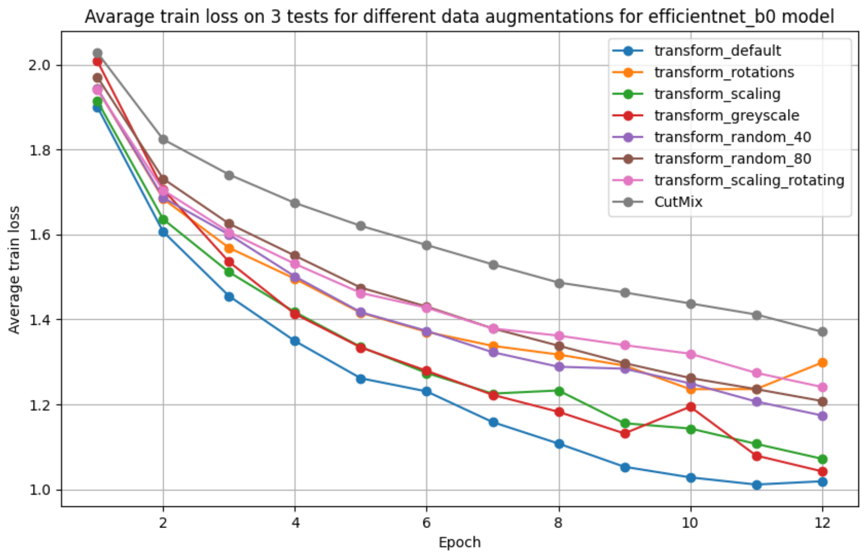


Figure 40: train loss - data augmentation with cutMix

What is interesting, CutMix outperformed other techniques. In just 12 epochs it get almost 70% accuracy on average. The loss may be similiar, as in training process it has to predict 2 classes at the same time, but accuracies are showing its domination. Scallng also showed it is an interesting and valueable method, while others are comparable.

We want to try this method on custom CNN 2, which is know trained with CutMix pictures. We give the model 60 epochs and we stop it only if 4 times in a row validation loss increase. The results are on the next diagram:

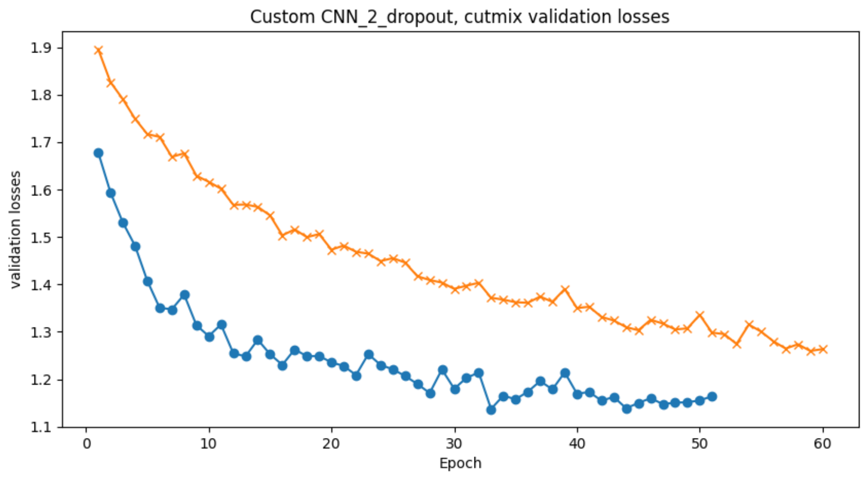


Figure 42: Cutmix custom CNN2, validation loss

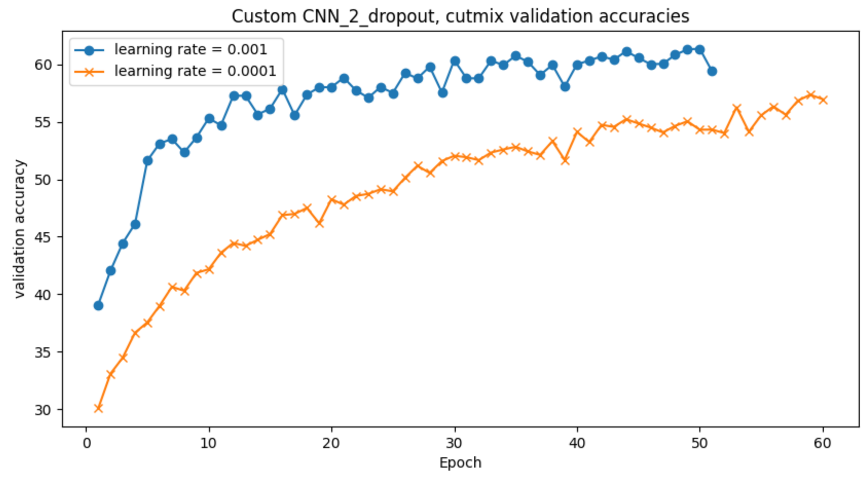


Figure 43: Cutmix custom CNN2, validation accuracy

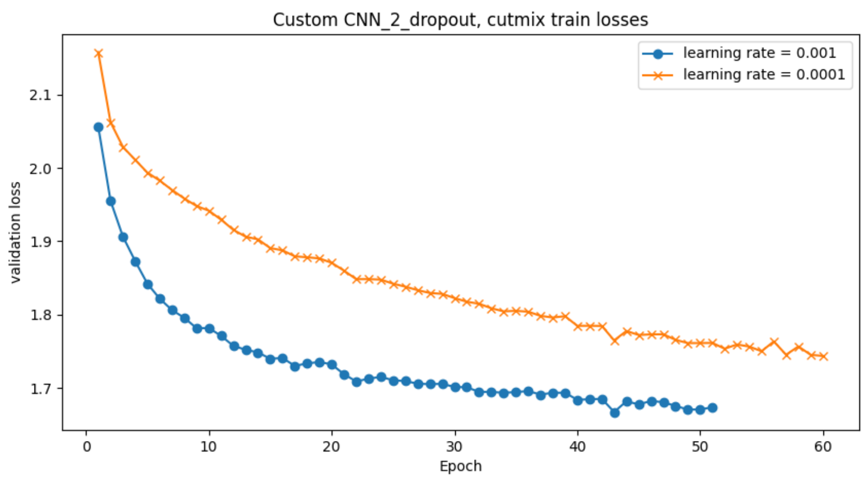


Figure 44: Cutmix custom CNN2, train loss

Even though the experiment was conducted, the models didn't achieve results as good, as before(around 63% vs around 61% now). The conclucion is, that sophisticated methods like CutMix works better on complex architectures, while on simple architectures, simple data augmentation works perfectly fine.

5 Prototypical networks - Fewshot learning technique

The next experiment which can be found in `Project1 fewshot new.ipynb` file, was conducted on pretrained DenseNet121 (pretraining on ImageNet dataset). We utilize a 30-shot 10-way technique using a prototypical network, which was introduced in the paper *Prototypical Networks for Few-shot Learning* by Snell. The model is being trained on 3000 episodes. The process begins by sampling small support and query sets from the dataset, where the support set contains a few labeled examples of each class (30-shot examples per 10-way classes), and the query set (50 images) contains additional examples to classify. The model acts as a feature extractor, generating embeddings for all images.

For each episode during training, the support set embeddings are averaged to create prototype vectors representing each class. The query set embeddings are then compared to these prototypes using Euclidean distance, producing a distance matrix. This matrix is converted into class probabilities via softmax, where smaller distances correspond to higher probabilities for the respective class. The training objective minimizes the negative log-likelihood of the correct class assignments, effectively teaching the model to map query images closer to their correct prototypes in the embedding space.

The validation and test phases follow the same procedure but without gradient updates. Performance is measured by classification accuracy on the query sets across multiple episodes. The optimizer, often AdamW, adjusts the model's weights to improve embedding quality over many episodes. The training of the models on Google Colab's GPU took approximately
$3\text{mins} \cdot 30 = 90\text{mins}$.

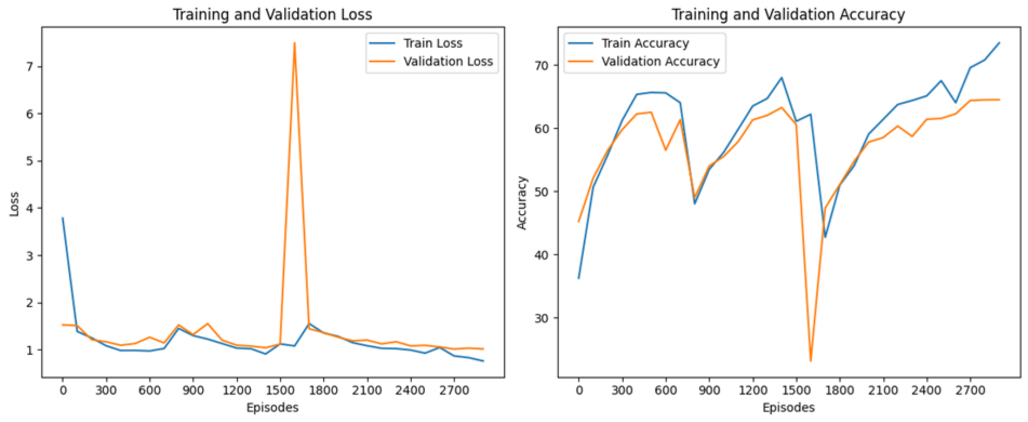


Figure 45: Training and validate metrics

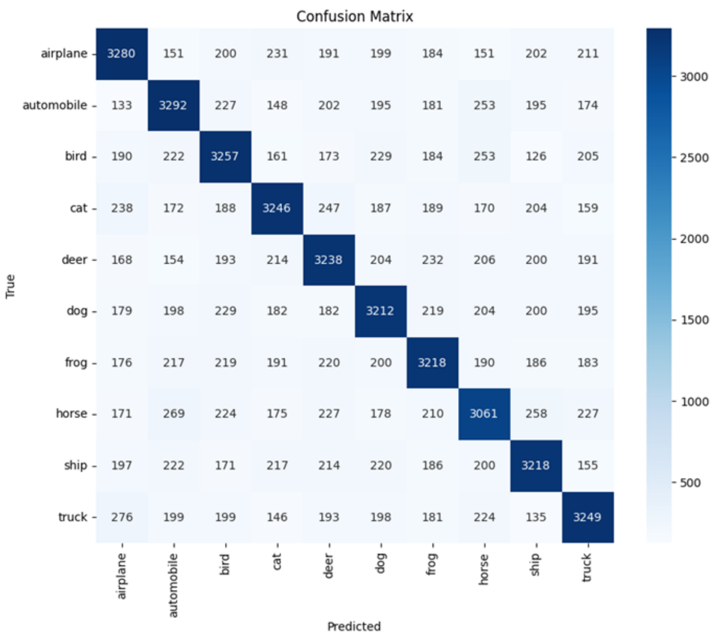


Figure 46: Confusion Matrix

Final Test Loss: 1.0198, Test Accuracy: 64.54%

We see that the model performance quite well, achieving good accuracy. We see, that there is a sudden spike in validation loss. It was checked, that the gradient vanishing was a problem, where its euclidean norm was near zero. There was no evidence of gradient exploding. What is also worth mentioning, is that in contrary to the other models, this models performs spectacularly with distincting "dogs" and other classes. The classes appear more distinguishable because the model is explicitly trained to maximize separability between class representations in a shared embedding space (e.g. during training, the model learns that "dog" prototypes should be far from "cat" prototypes in Euclidean space, even if their raw pixels are similar).

5.1 Impact on reduced dataset

The next experiment that we want to conduct is the impact of reducing the dataset of the model from 90 000 images in training data to 30 000 images. All the previous assumptions about the model stays the same.

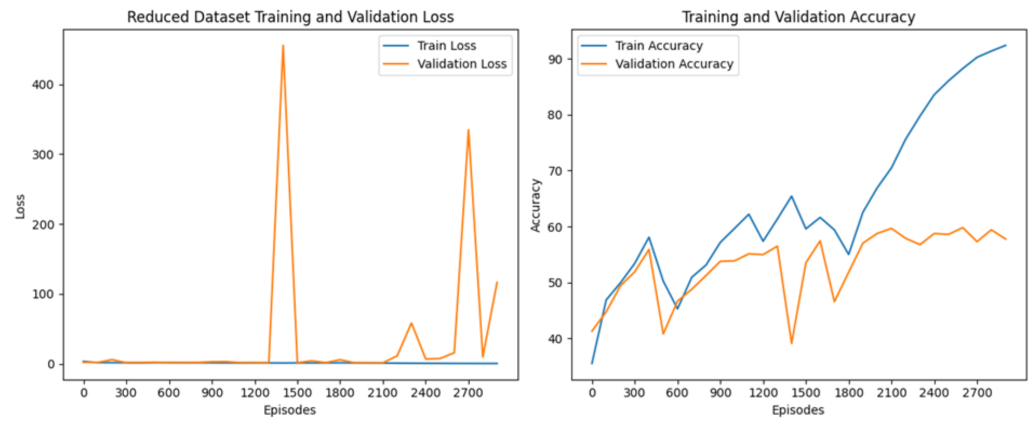
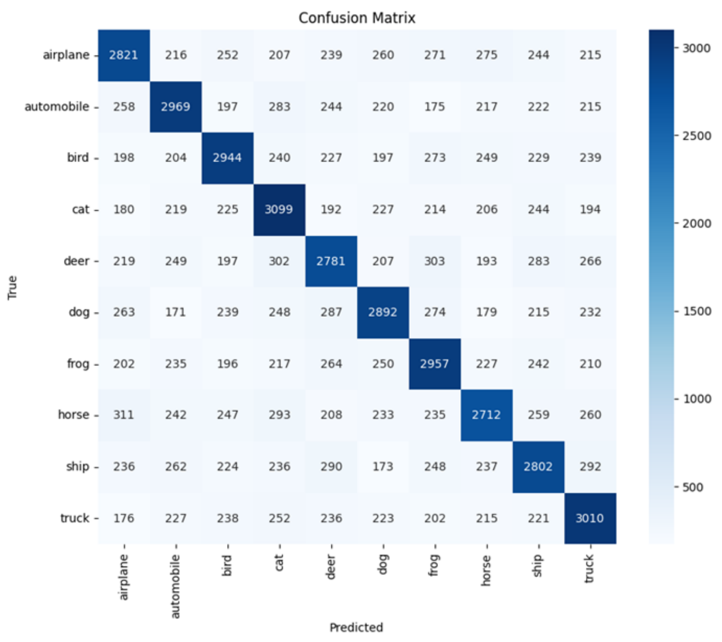


Figure 47: Training and validate metrics on reduced dataset



	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	2821	216	252	207	239	260	271	275	244	215
automobile	258	2969	197	283	244	220	175	217	222	215
bird	198	204	2944	240	227	197	273	249	229	239
cat	180	219	225	3099	192	227	214	206	244	194
deer	219	249	197	302	2781	207	303	193	283	266
dog	263	171	239	248	287	2892	274	179	215	232
frog	202	235	196	217	264	250	2957	227	242	210
horse	311	242	247	293	208	233	235	2712	259	260
ship	236	262	224	236	290	173	248	237	2802	292
truck	176	227	238	252	236	223	202	215	221	3010

Figure 48: Confusion Matrix on reduced dataset

Reduced Test Loss: 226.42, Test Accuracy: 57.97%

As previously, the model performance, despite reduced dataset has satisfying accuracy. We see, that the loss becomes bigger, which also would be due to the vanishing gradient. Once again, the model correctly recognizes the "dog" class, which was not the case in previous models.

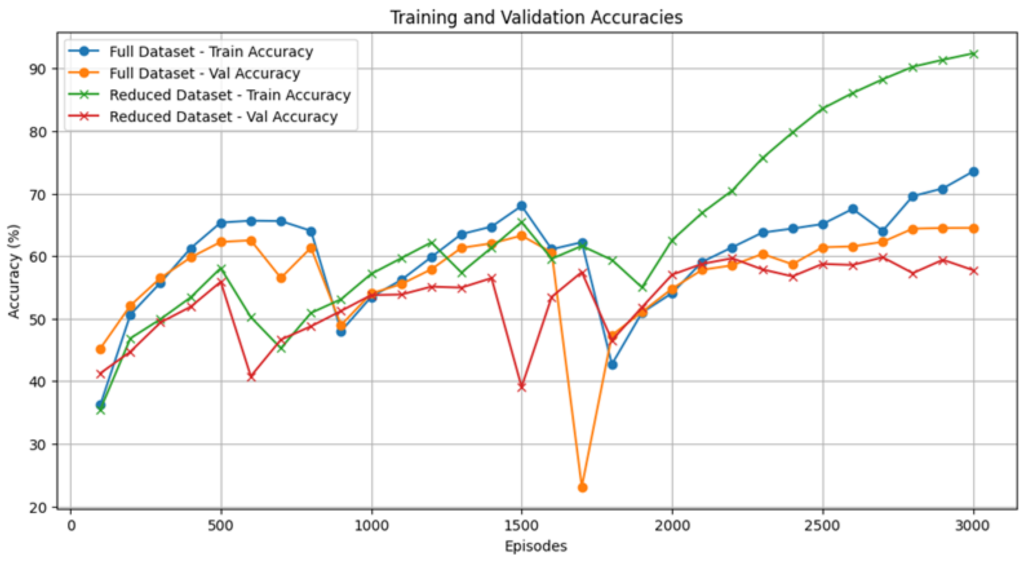


Figure 49: Training and validate metrics on reduced dataset

We also see, that the model trained on reduced dataset is being overfitted, especially after the 2000th episode, which is not the case with the model trained on the full dataset. Lastly, we compare the resulted accuracies on test set.

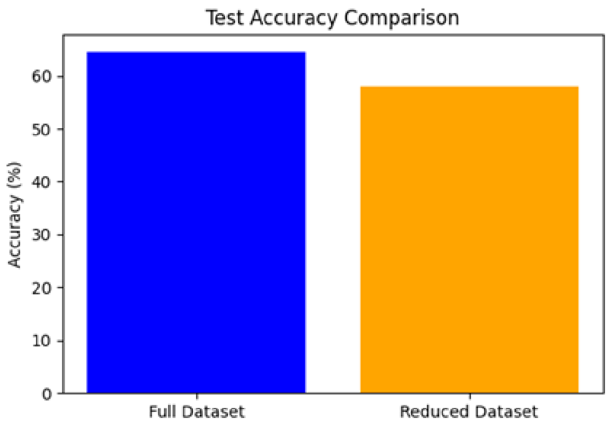


Figure 50: Confusion Matrix on reduced dataset

6 References

Prototypical Networks for Few-shot Learning by Snell et al. (2017), in *Advances in Neural Information Processing Systems* (NeurIPS), Vol. 30, pp. 4077–4087.

EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks(2019) by Mingxing Tan, Quoc V. Le