FINAL REPORT

DEEP LEARNING – Convolutional neural networks

Authors: Jan Wojtas, Mikołaj Zalewski

Faculty of Mathematics and Information Science

Warsaw University Of Technology

March 2023

Project description

The main goal of this project was to perform image classification on *Cifar10* dataset using convolutional neural networks and understanding how CNNs work in practice.

More explicitly, the most emphasis was placed on:

- Designing different CNN architectures.
- Comparing our architectures with existing ones and other, pre-trained models.
- Investigating influence of hyperparameter tuning on accuracy and overall performance of networks.
- Investigating influence of applying different augmentation techniques on dataset. Experiment with various types of augmentation, from basic to more advanced.
- Investigating quality of prediction with ensemble application.

Project environment

1. Chosen programming language: **Python**.

2. The whole project (including preparing datasets, designing Convolutional Neural Networks and training) is based on **PyTorch** - powerful Python framework for Deep Learning.

3. We used GitHub repository for file hosting and versioning our project. Link to the repository can be found here: https://github.com/mikolajzalewski/Deep_Learning

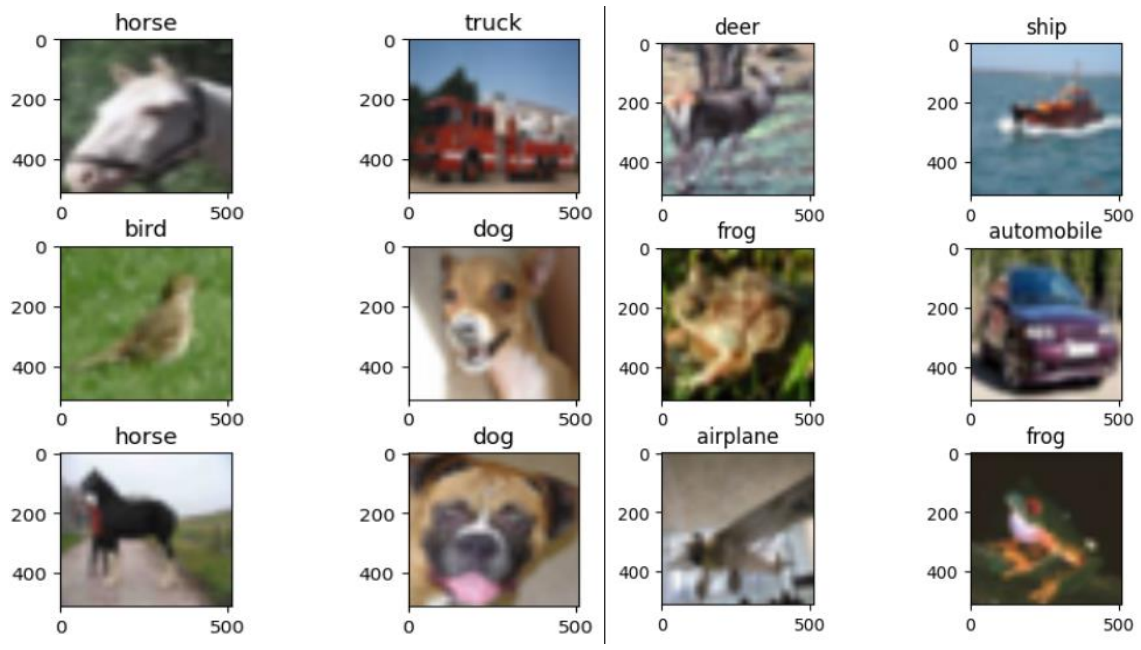
4. Our project files are divided into ones with **.py** extension (for classes, functions, global variables) and those with **.ipynb** extension (for visualization of results and data, tuning and training neural networks). The Idea was to keep many, simple notebooks for computations and store most of the base code in **.py** files - to make it easy to understand and keep the clarity.

Cifar10 dataset

1. Overview

In this project we worked on *Cifar-10 dataset*. The training data consists of 50 000 images, where each image belongs to one of 10 possible classes: *'frog'*, '*truck*', *'deer*', '*automobile*', '*bird*', '*horse*', *'ship'*, '*cat'*, '*dog*', '*airplane*'.
There are 5 000 images per each class.

Each image's size is 32 x 32 pixels and each image has 3 RGB color channels. A sample of images from Cifar10 dataset can be seen on the picture below (each image has been resized to 512 x 512 using bilinear interpolation to give it more sharpness):



2. Data preparation

Firstly, we downloaded *Cifar10* data from Kaggle and the PyTorch Dataset was prepared to keep our images in structured way. PyTorch Dataset is a convenient way to store the data, as during network training it allows us to gather the images directly from the folder where the images are kept, instead of copying all the data to a new variable. Moreover, PyTorch Dataset is compatible with PyTorch DataLoader class, which eventually is needed to pass batches of our dataset for training of the CNN model.

Our goal is to tune our Convolutional Neural Network models to perform the best possible classification of the Cifar10 images. Therefore, we need to have a portion of data devoted to validation of our models. We considered two main options: k-Fold cross-validation and standard train-validation split.

The advantage of k-Fold cross-validation is that we use all of our data for both training and validation, while by using classic train-validation split we decrease the size of our training data. On the other hand, k–Fold cross-validation is much more time-costly.

As training deep neural networks is a very long and time-consuming procedure, we decided to stick to train-validation split (with proportion 80%-20%) and we created another pyTorch Dataset for validation data.

3. Data augmentation

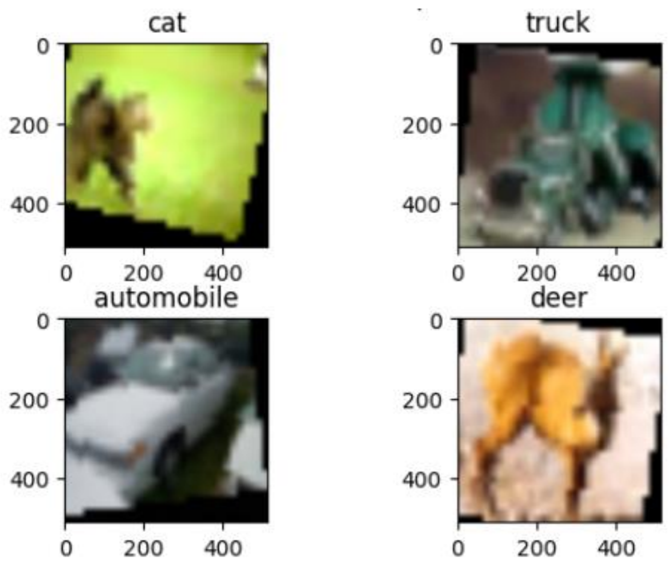
Data augmentation is a very popular technique to artificially enlarge the training data with new images and reduce overfitting of the model on training data. It is achieved by applying different transformations to the training dataset [1], including Geometric Transformations (Rotation, Flip), Photometric Transformations (Color Jitter, Edge enhancement) along with more advanced techniques (Cutout, Mixup). We decided to test 3 different augmentation techniques and verify its impact on models' accuracies.

Basic augmentation

We tested out multiple simple augmentation techniques combined together, including:

- Random rotation – which is rotation of the picture by random angle from specified range. We chose range (-30°, 30°), as such rotation is logical with respect to our dataset. We would like to point out, that for example rotation by 180° of 'ship' image would not make much sense – at least from human point of view.
- Color Jitter – which is a manipulation of brightness and contrast of the images
- Random Crop – which is a cropping of the image at randomly chosen spot with given size. We decided to add 4 pixels of padding to each image and then crop the images at random with size equal to 32. Other option to be considered would be to apply smaller crop size and then resize the image to 32 x 32 size.
- Gaussian Noise – which is adding gaussian noise to each channel of image with given mean and variance. We decided to keep it relatively small (mean=0, std=0.001), just to introduce some noise but not at the level, that the images would not be recognizable.
- Random flip – which is a random horizontal flip of an image, with probability of p = 0.5. (as mentioned before, vertical flip would not make much sense)

A sample of images after applying basic augmentation can be seen on the figure below:



Mixup

Mixup is an interesting and advanced technique of image augmentation. The idea behind this technique is to mix pairs of images together by averaging their pixel values from each RGB channel (could be weighted average) and give the resulting image label of the first mixed image. A very interesting note on mixing images from *Cifar10* dataset can be found in [1]. First of all, it was reported that this augmentation technique (applied together with Random Flip and Random Crop) gave a drop in error rate on *CIfar10* from 8.22% to 6.63 %. It was supposed to work incredibly well on reduced data (when only 1000 of total images with 100 images per class were considered) where it gave a drop from 43.1 to 31.0% in error rate.

The most interesting thing about mixup mentioned in the article is that it worked the best, when mixing images from all classes (not just from the same class), which seems very counterintuitive on the first sight.

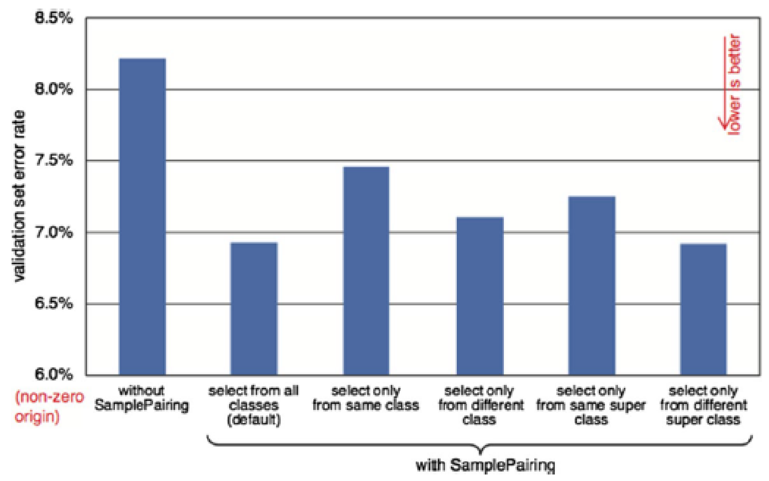
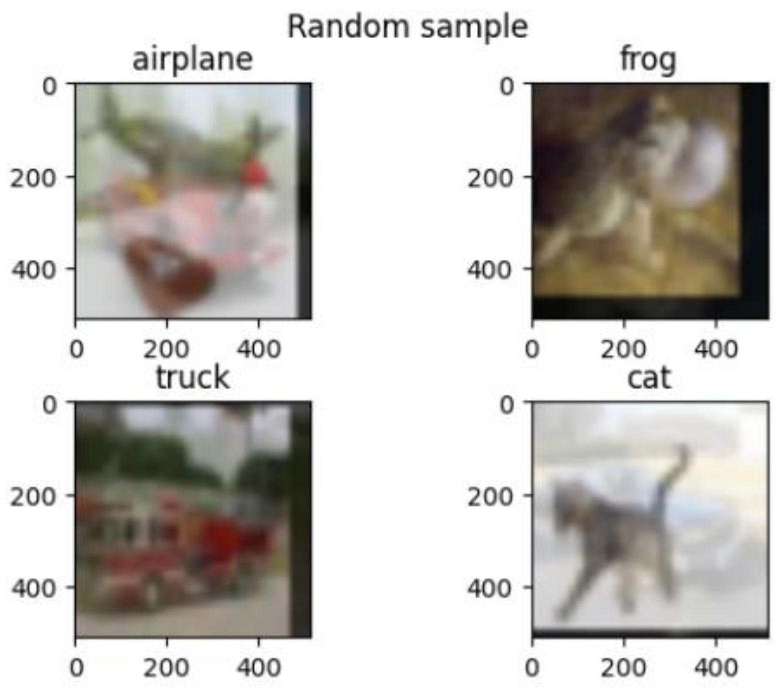


Fig. 8 Results on the reduced CIFAR-10 dataset. Experimental results demonstrated with respect to sampling pools for image mixing [65]

Source: [1]

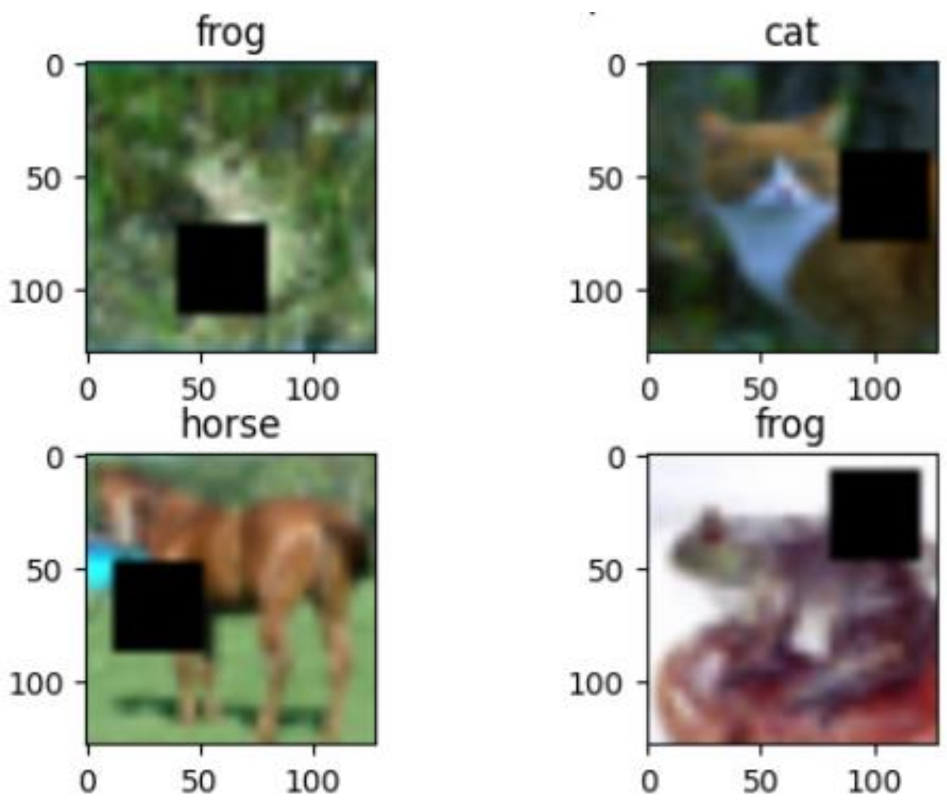
Therefore, we tested out the mixup of images from different classes and mixed images with proportion of 0.7. This means, that if we took two images and mixed them together, the first one of them would have weight 0.7 and the second one 0.3. Next, we would give label to our mixed image, being the label of the image with more weight. Sample of the mixed dataset can be seen on the figure below:



We can see, that for example in first picture airplane was mixed with some kind of vehicle (‘truck’ or ‘car’) in proportion (70%-30%).

Cutout

Cutout is another advanced technique, common in data augmentation world. Its main idea is to select a rectangle of size n x m from an image and replace all pixel values inside rectangle with either 0 or 255. In [1] it is reported to give a reduction of error from 5.17% to 4.31 % on CIfar10 dataset. In our project, we implemented method which cuts out rectangle of size 10 x 10 pixels and replaces their values with 255 in all RGB channels. The result of such augmentation can be see below:



Convolutional Neural Networks - design

In this chapter we review the architecture of 3 convolutional neural networks, that we used for performing classification on *Cifar10*.

1. CNN_3_class

We discuss the architecture of a CNN called **CNN_3_class**, which is designed to classify images from the CIFAR-10 dataset. We decided to keep the number and type of layers stable, while considering multiple setups of other parameters (number of filters, neurons etc.) and tuning the architecture to achieve the highest possible accuracy.

The structure of the CNN_3_class network consists of several layers, each with its own unique properties and functionality. The order and type of layers were chosen based on an article [6].

The first layer of the network is a convolutional layer, which applies filters to the input image to extract features. The Conv2d function in PyTorch library is used to create this layer. This layer has 3 input channels (RGB images) and 32 output channels (number_of_filters0) with a kernel size of 3 (kernel_size1), stride of 1, and padding of 1. The padding helps in preserving the spatial dimensions of the input image after convolution. After the convolution, the ReLU activation function is applied to introduce non-linearity.

The next layer in the CNN_3_class network is a max-pooling layer, which reduces the spatial size of the output from the previous convolutional layer while retaining important features. The MaxPool2d function in PyTorch library is used to create this layer with a pool size of 2. This layer is followed by another convolutional layer with 32 input channels and 256 output channels (number_of_filters1), a kernel size of 3 (kernel_size2), stride of 1, and padding of 1. Similar to the previous convolutional layer, ReLU activation function is applied after the convolution.

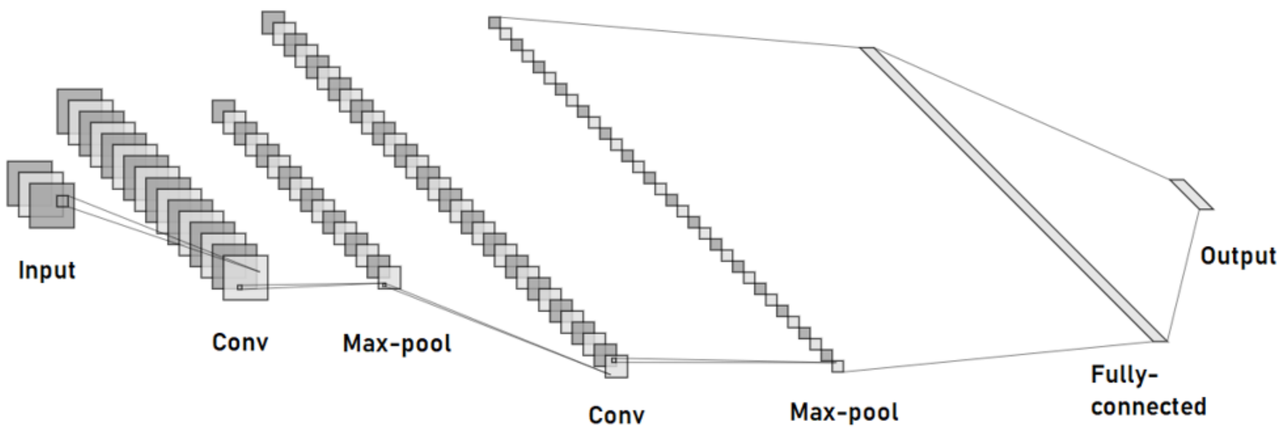
The next layer in the CNN_3_class network is another convolutional layer, which applies filters to the output from the previous max-pooling layer. This layer has 32 input channels (same as the output channels of the previous convolutional layer) and 256 output channels (number_of_filters1) with a kernel size of 3 (kernel_size2), stride of 1, and padding of 1. After the convolution, the ReLU activation function is applied to introduce non-linearity.

The next layer is again a max-pooling layer that reduces the spatial size of the output from the previous convolutional layer while retaining important features. This layer uses a kernel size of 2 and no padding. This results in the output size being half of the input size, i.e., the length of the input image divided by 4.

The next layer is a fully connected layer, which takes the flattened output from the previous max-pooling layer and passes it through a set of neurons. This layer has *no_neurons* (by default *no_neurons* = 500) number of neurons, and the ReLU activation function is applied to introduce non-linearity.

The final layer in the CNN_3_class network is another fully connected layer, which takes the output from the previous fully connected layer and passes it through a set of neurons. This layer has num_classes (10 in this case, as we are classifying images from the CIFAR-10 dataset) number of neurons, and no activation function is applied.

General schema of the network can be seen on the picture below:

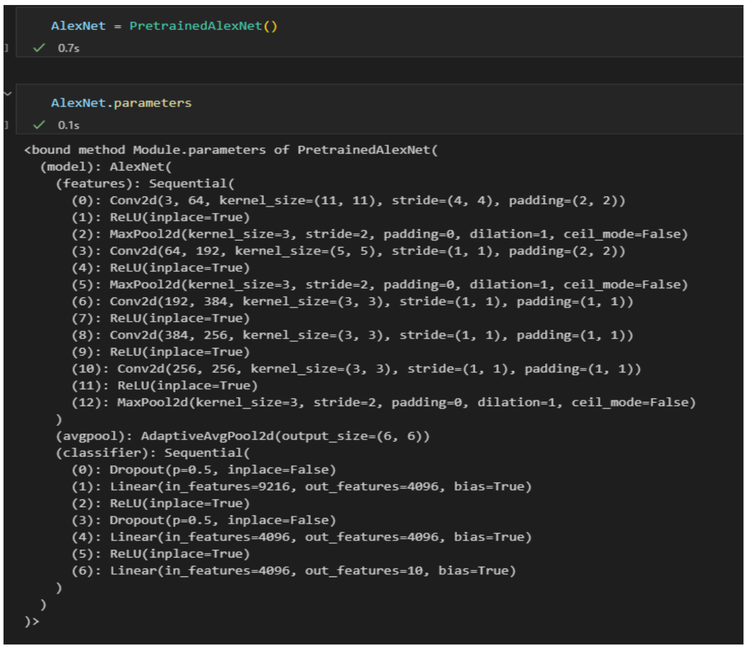


2. PretrainedAlexNet

The PretrainedAlexNet network is a convolutional neural network architecture that is based on the AlexNet model, which was the winner of the 2012 ImageNet Large Scale Visual Recognition Challenge. It is pretrained on ImageNet data.

The architecture is as follows:

- The input images are of size 3x224x224, representing a 224x224 RGB image.
- The first layer is a convolutional layer with 64 filters of size 11x11, a stride of 4 and padding of 2.
- The second layer is a ReLU activation function.
- The third layer is a max pooling layer with a kernel size of 3 and a stride of 2.
- The fourth layer is a convolutional layer with 192 filters of size 5x5 and padding of 2.
- The fifth layer is a ReLU activation function.
- The sixth layer is a max pooling layer with a kernel size of 3 and a stride of 2.
- The seventh layer is a convolutional layer with 384 filters of size 3x3 and padding of 1.
- The eighth layer is a ReLU activation function.
- The ninth layer is a convolutional layer with 256 filters of size 3x3 and padding of 1.
- The tenth layer is a ReLU activation function.
- The eleventh layer is a convolutional layer with 256 filters of size 3x3 and padding of 1.
- The twelfth layer is a max pooling layer with a kernel size of 3 and a stride of 2.
- The thirteenth layer is a fully connected layer with 4096 neurons.
- The fourteenth layer is a ReLU activation function.
- The fifteenth layer is a dropout layer with a dropout rate of 0.5.
- The sixteenth layer is a fully connected layer with 4096 neurons.
- The seventeenth layer is a ReLU activation function.
- The eighteenth layer is a dropout layer with a dropout rate of 0.5.
- The nineteenth layer is a fully connected layer with 10 neurons, which is the number of classes in the CIFAR-10 dataset.



```
AlexNet = PretrainedAlexNet()
```
✓ 0.7s

```
AlexNet.parameters
```
✓ 0.1s

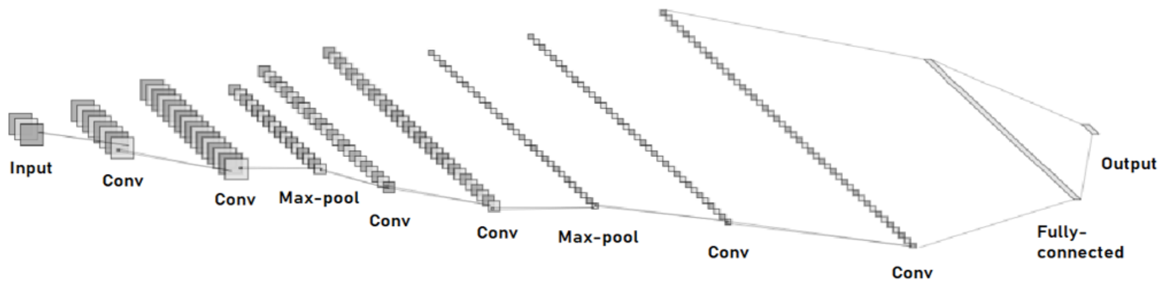
```
<bound method Module.parameters of PretrainedAlexNet(
  (model): AlexNet(
    (features): Sequential(
      (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
      (1): ReLU(inplace=True)
      (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
      (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
      (4): ReLU(inplace=True)
      (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
      (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (7): ReLU(inplace=True)
      (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (9): ReLU(inplace=True)
      (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (11): ReLU(inplace=True)
      (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
    (classifier): Sequential(
      (0): Dropout(p=0.5, inplace=False)
      (1): Linear(in_features=9216, out_features=4096, bias=True)
      (2): ReLU(inplace=True)
      (3): Dropout(p=0.5, inplace=False)
      (4): Linear(in_features=4096, out_features=4096, bias=True)
      (5): ReLU(inplace=True)
      (6): Linear(in_features=4096, out_features=10, bias=True)
    )
  )
)>
```

AlexNet was trained on a specific dataset, and therefore, when preparing the datasets for training and prediction using this model, specific transformations such as resizing and normalization were necessary. The transformations applied to the dataset, which include resizing the image to a 256x256 size using the transforms.Resize method, converting the image to a tensor using transforms.ToTensor, and normalizing the image using transforms.Normalize with a mean of [0.485, 0.456, 0.406] and a standard deviation of [0.229, 0.224, 0.225]. These transformations are crucial in ensuring that the input data is in the correct format and range expected by the AlexNet model.

The PretrainedAlexNet is a modification of the original AlexNet architecture in which the last fully connected layer has been replaced with a new one to output the number of classes in the CIFAR-10 dataset, which is 10.

3. MyCNN

MyCNN is a model based on results from [2]. Its number of convolutional layers and fully connected, dense layers was determined by weighted random search optimization algorithm described in aforementioned article. The model which achieved highest accuracy had 6 convolutional layers and one fully-connected layers with C1 = 736, C2 = 508, C3 = 664, C4 = 916, C5 = 186, C6 352 and F1 = 1229. C_n Is number of filters in n-th convolutional layer while F1 is number of neurons In fully-connected layer. Schema of this network can be seen below:



4. Ensemble

Ensemble methods are machine learning techniques that combine multiple models to improve predictive performance. The idea is to leverage the strengths of individual models by aggregating their predictions. In this case, we have three models - model1, model2, and model3 - trained on the training set, and we will be using them to build two ensemble models: model_mean and model_max. These models will be tested on the validation set, which contains 10,000 observations.

The reason we use ensemble methods is that they can improve the accuracy and robustness of predictions by reducing the variance of individual models. Ensemble methods also help to reduce overfitting and increase generalization by combining multiple models with different biases.

The two specific ensemble methods used in this code are model_mean and model_max. Model_mean calculates the average prediction of the three models, while model_max selects the highest prediction among the three models. The advantage of model_mean is that it should provides a more stable prediction by reducing the impact of outliers. On the other hand, model_max should be more aggressive and may perform better when the models have complementary strengths.

Hyperparameter tuning

Tuning methods

There are different, popular methods of hyperparameter tuning. Among all of them, we decided to pay special attention to grid search, random search and novel but powerful method – weighted random search [2]. The implementation of these methods can be found in our project files.

Grid Search

Most common and oldest technique for searching for optimal hyperparameters. It involves creating a parameter grid and searching, one-by-one, each parameter combination from the aforementioned grid. In case of deep learning, this method is very time consuming as there are a lot of parameters to tune (including parameters for optimizers and network architecture). For example, if we wanted to tune only 10 hyperparameters and for each check 4 possible values, we would get 4^10 training processes which is nearly impossible to perform (as each training of deep neural network could last for hours or days). Therefore, in our project, we mostly considered `simplified` version of grid search - a `greedy algorithm`. We tune the parameters step by step, meaning that we tune first parameter alone, then tune the second parameter with the best first parameter possible etc. It greatly reduced the time complexity of training and allowed us to test more, various options.

Random Search

It is second most popular technique of tuning hyperparameters. It relies on choosing the parameter sets at random. It is supposed to be better than grid search when considering large number of trials, but as we lacked computing power and time, we decided to compare it with Grid Search on small number of trials, where it gave much worse results that our 'greedy' version of grid search.

Weighted Random Search

This method was introduced in [2] as improvement to random search. Its main idea is to run the random search N_0 times (which is relatively small, compared to the total number of searches) and using fANOVA algorithm determine which parameters are more and which less important with the respect to value of goal function (in our case accuracy of the model on validation data). Then we assign weights to each hyperparameter, corresponding to their importance and perform so called ‘weighted random search’ on the remaining searches. In [2] this method was reported to give highest accuracy on Cifar10 data with regard to other methods, while tuning network architecture parameters (number of layers and number of filters/neurons in each layer).

Source: [2]

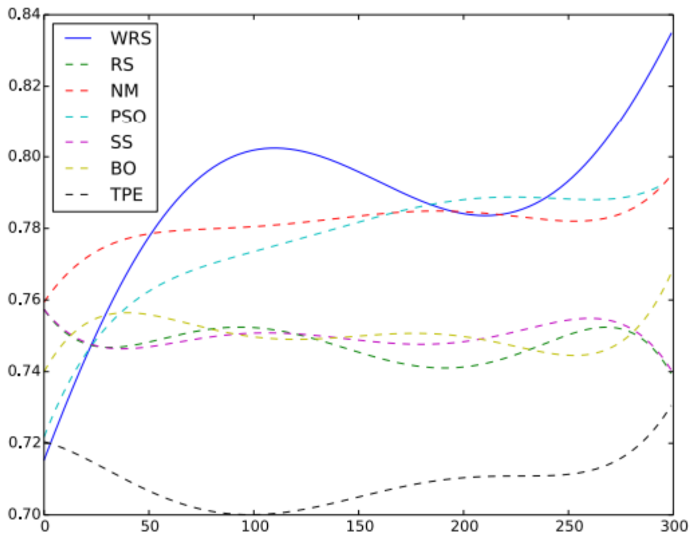


Figure 2: Least squares five degree polynomial fit on RS, NM, PSO, SS, BO, TPE vs. WRS accuracy for CIFAR-10 on 300 trials. The plot shows the values obtained at each iteration

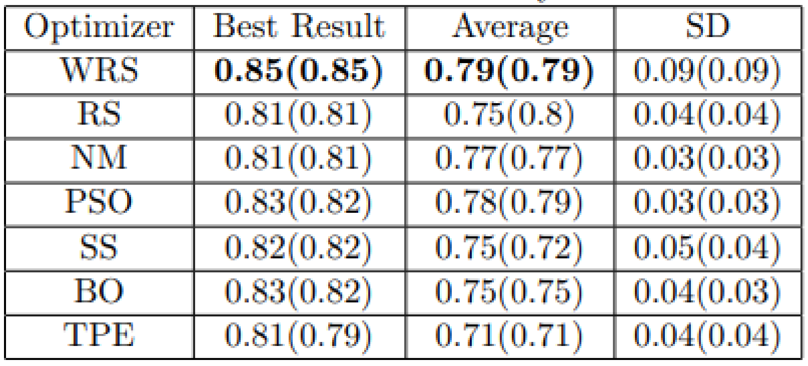


Table 2: Obtained CNN accuracy on CIFAR-10

Optimizer	Best Result	Average	SD
WRS	**0.85(0.85)**	**0.79(0.79)**	0.09(0.09)
RS	0.81(0.81)	0.75(0.8)	0.04(0.04)
NM	0.81(0.81)	0.77(0.77)	0.03(0.03)
PSO	0.83(0.82)	0.78(0.79)	0.03(0.03)
SS	0.82(0.82)	0.75(0.72)	0.05(0.04)
BO	0.83(0.82)	0.75(0.75)	0.04(0.03)
TPE	0.81(0.79)	0.71(0.71)	0.04(0.04)

Source: [2]

The algorithm is powerful, but its strength can be seen only after large number of trials (as we can see on the chart, it overcame other algorithms after about 250 trial). Therefore we didn’t try out this algorithm in our case, but we tried out the architecture of the network, which was tuned with weighted random search (and Is described in chapter about Convolutional Neural Networks).

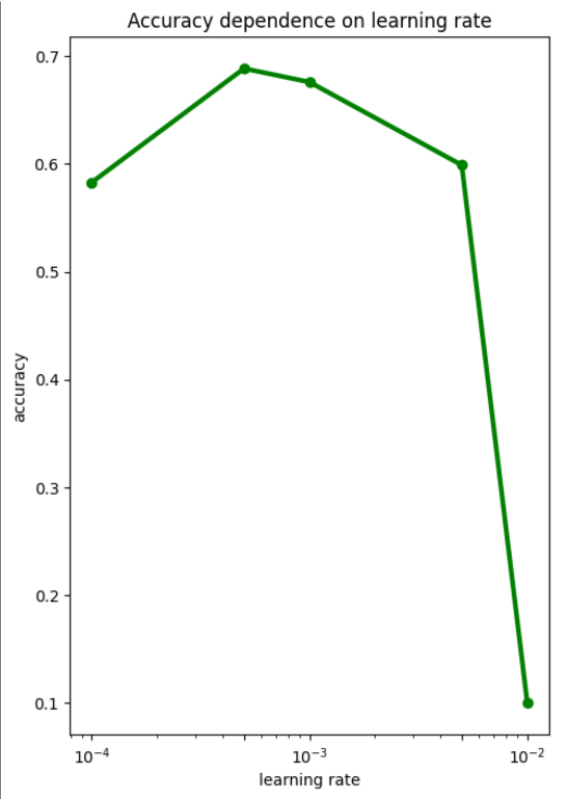
Tuning models

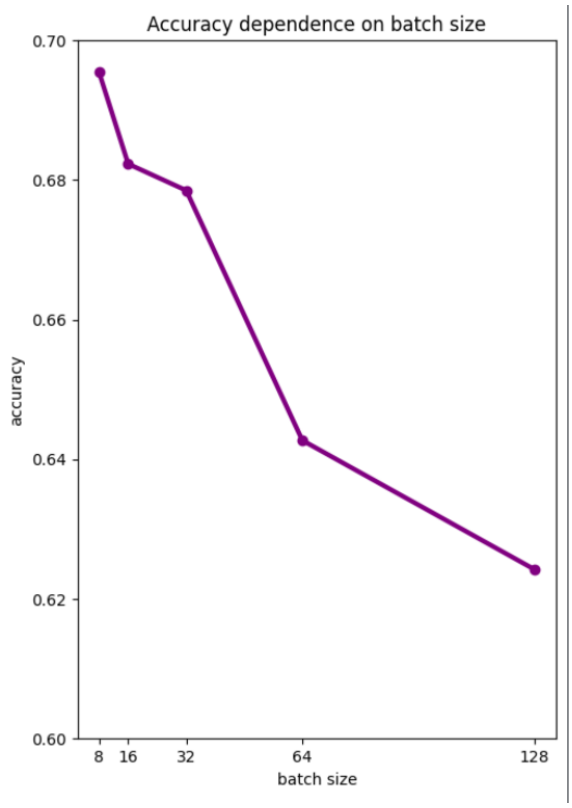
CNN_3_class

When it comes to tuning hyperparameters of our networks, we mostly focused on **CNN_3_class** network. As mentioned above, we decided to search for hyperparameters step-by-step in the following ordering: learning rate and batch size, dropout and weight regularization, architectural parameters (number of filters, number of neurons), number of epochs. We decided to consider accuracy metric when it comes to evaluation of models with different hyperparameters. All hyperparameters were tuned on 5 epochs.

Learning rate and batch size

These 2 parameters are very important and thus we decided to tune them at first.

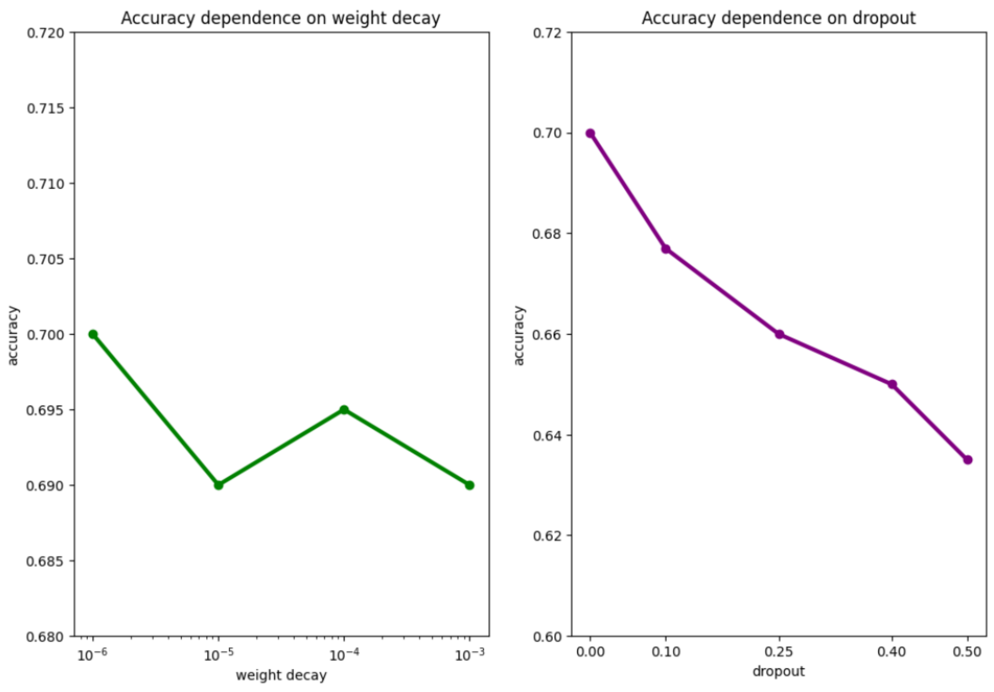




Learning rate of 0.0001 seemed to approach the minimum of our goal function too slowly while with 0.01 the optimization algorithm probably took too big steps and would not converge. The optimal one was somewhere in the middle (0.0005).

The optimal batch size was the smallest one, which makes sense, as Cifar10 dataset has only 10 classes. Therefore the model does not need to see too many images in every single batch to perform well.

Dropout and weight decay

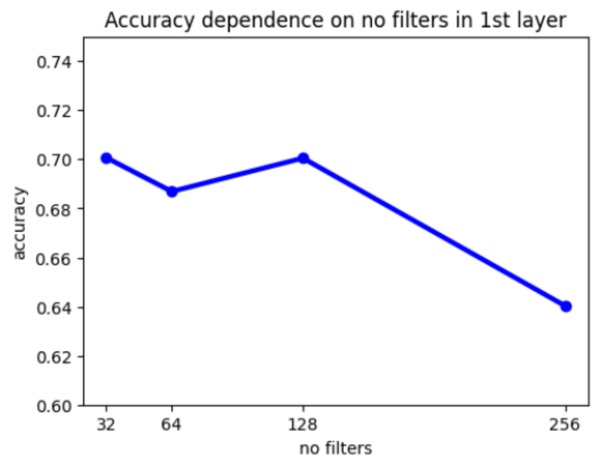


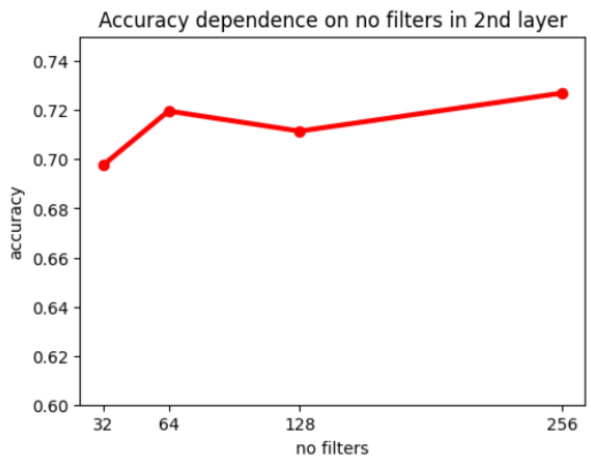
Both dropout and weight decay are tools to control overfitting of the model to training data. In our case, they did not improve accuracy of the model. We suppose that it is due to the fact, that all parameters were tuned on 5 epochs, which might be too short time for our network to overfit.

Number of filters

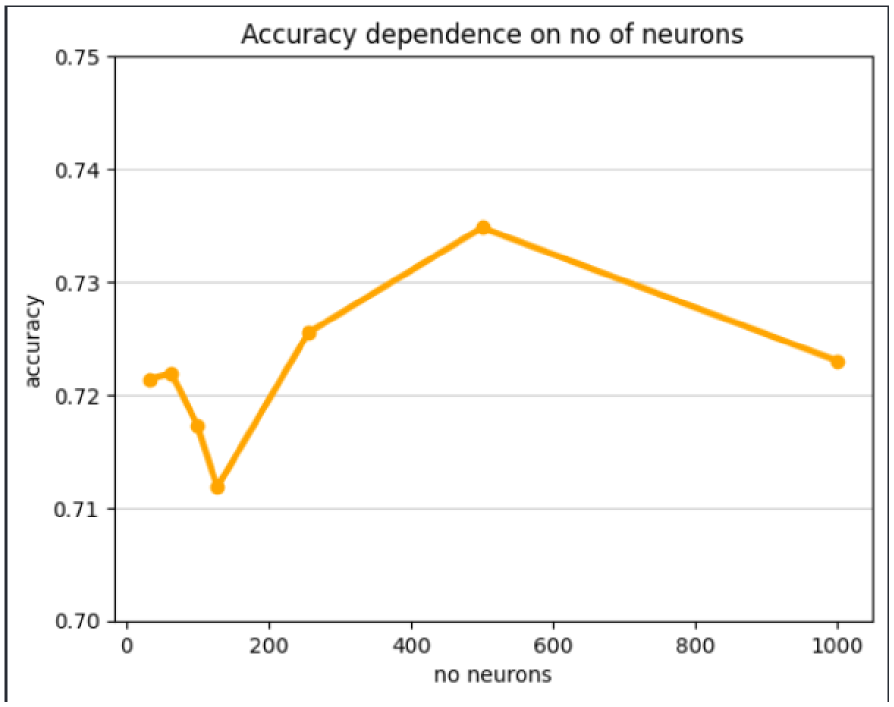
CNN_3_class model has two convolutional layers, so we tuned the number of filters in each one of them.

We found out that the most influential was number of filters in second layer. The first layer captures more general features, while second one more detailed ones. Therefore presumably our model needed more filters in second layer to increase precision in distinction between different classes based on details.





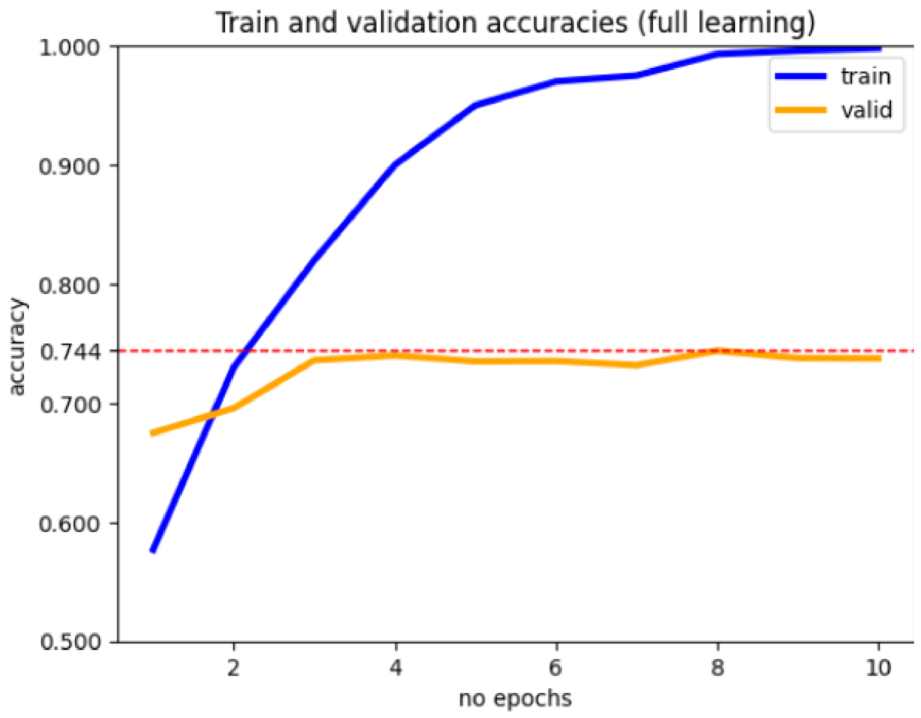
Number of neurons

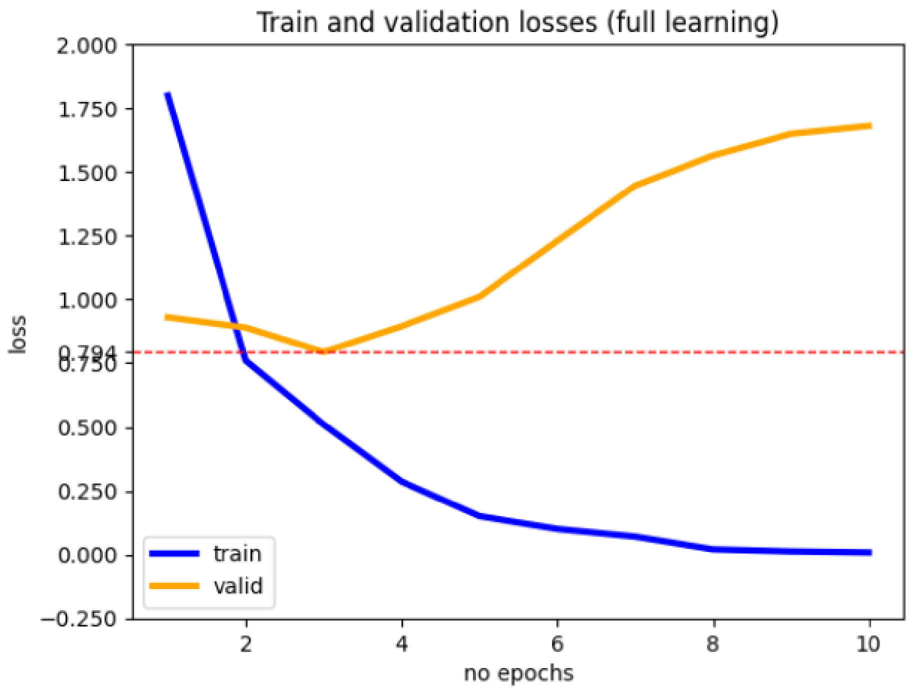


As number of filters in second layer increased greatly (from 32 to 256), the fully connected layer also preferred to have more neurons. After initial experiment of checking the number of neurons up to 500, we noticed substantial increase of accuracy. It encouraged us to perform test for even bigger number of neurons (1000), but the accuracy decreased. Therefore we kept 500 neurons in fully-connected layer.

Number of epochs

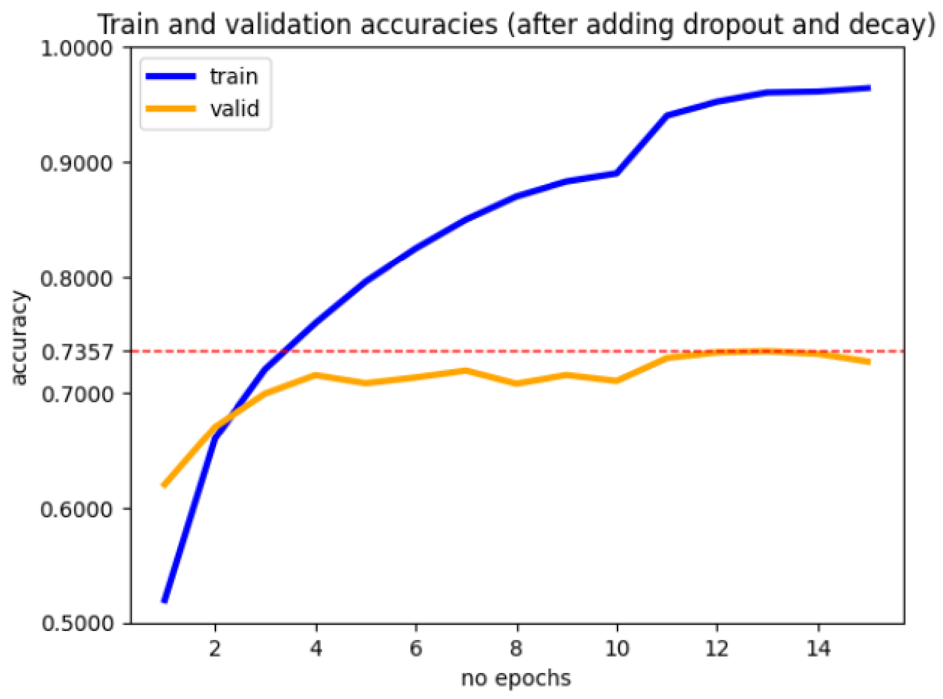
We decided to run our model on 10 epochs and check when the accuracy starts to decrease and loss starts to increase.

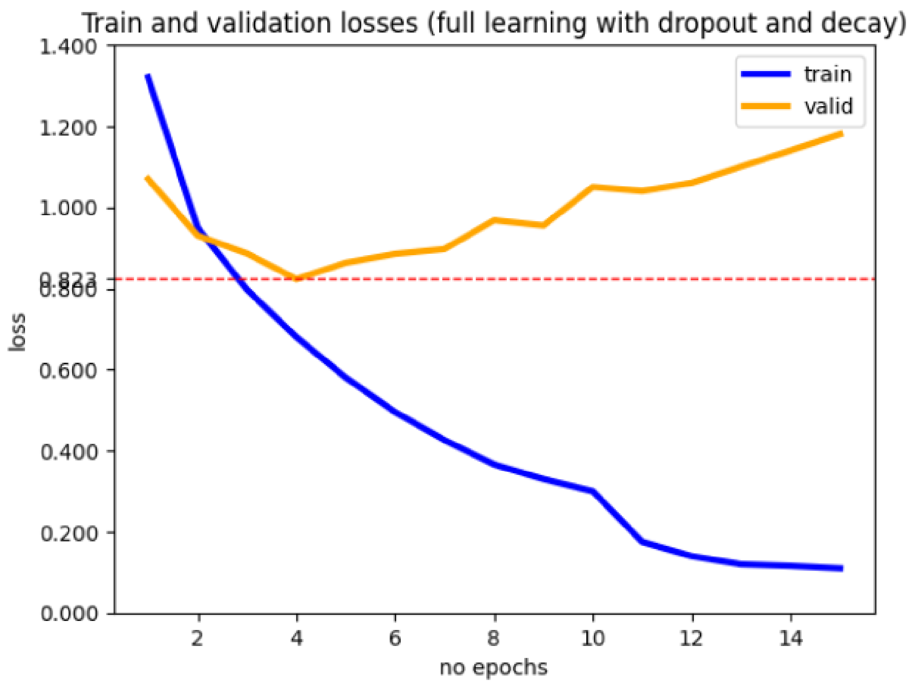




As we can notice, the accuracy of model is at its peak at 8th epoch, but the loss starts to increase already from 3rd epoch. Considering that at 3rd epoch the accuracy is only slightly lower than after 8th epoch we decided that three epochs is enough, yielding approximately 74,4% of accuracy.

As mentioned before, dropout and weight decay did not give good results after 5 epochs, so we decided to check their importance once more, but on higher number of epochs.

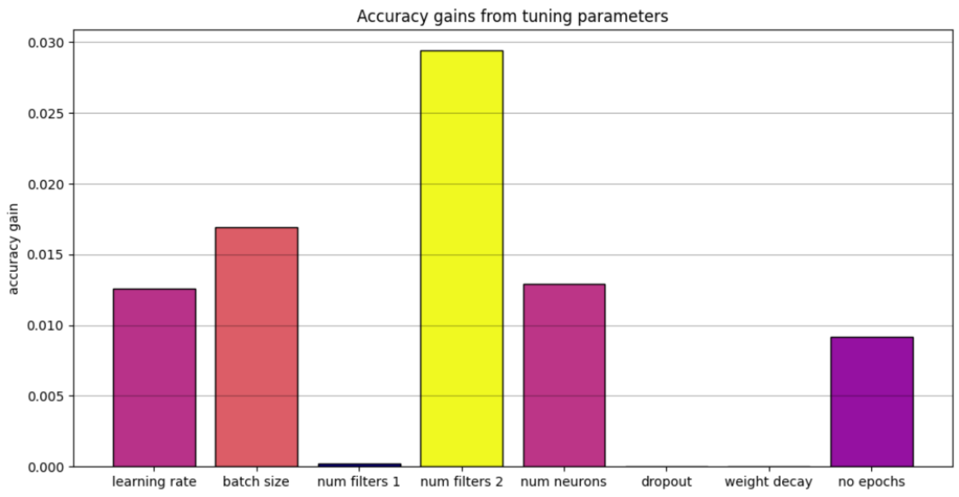




We can see that the models does not overfit as fast as without dropout and decay, but the overall performance does not improve.

Final results

We summarize the results of hyperparameter tuning on the chart below:



The most important parameters in case of CNN_3_class model seem to be learning rate, batch size and number of filters in second layer.

AlexNet

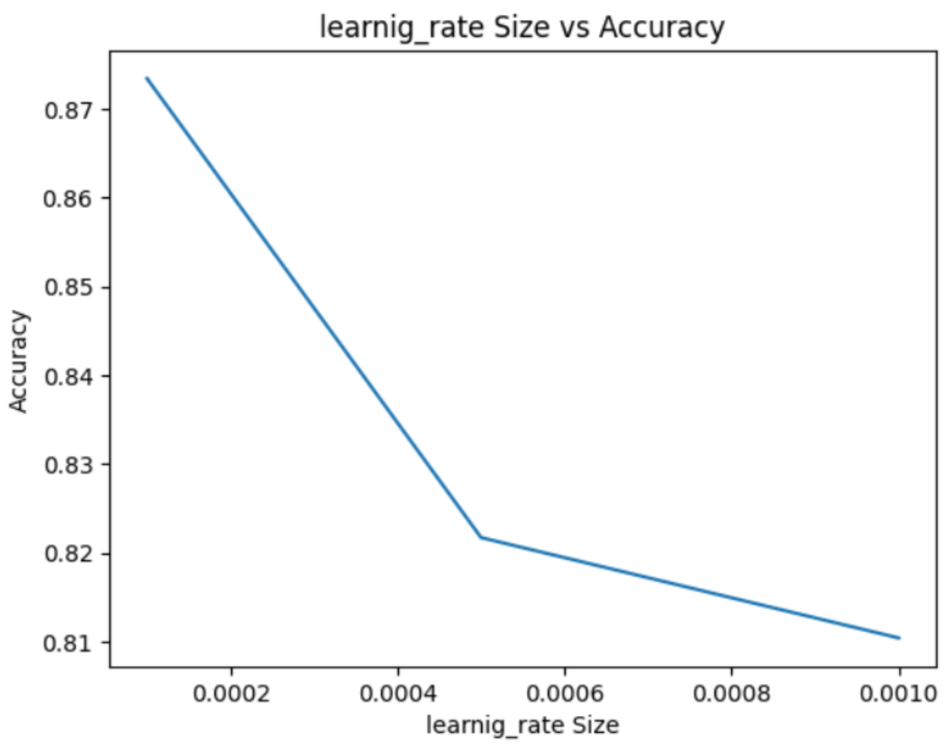
In this part, we focused on tuning two important hyperparameters: batch size and learning rate. As the network architecture was fixed, we didn't change e.g. the number of neurons in the layers.

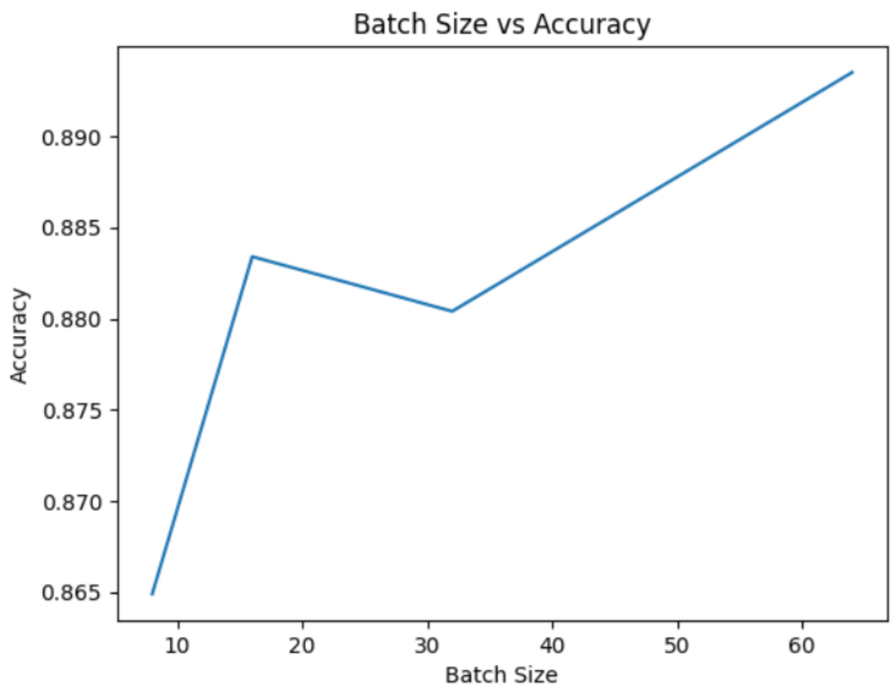
Batch size and Learning rate

Batch size refers to the number of training examples utilized in one iteration of gradient descent. A larger batch size can lead to faster convergence but requires more memory to store intermediate computations. On the other hand, a smaller batch size may take longer to converge but allows for better generalization.

Learning rate is a hyperparameter that controls the step size at each iteration during gradient descent. A high learning rate can cause the model to overshoot the optimal solution, while a low learning rate can result in slow convergence or getting stuck in a local minimum. Finding an optimal learning rate is crucial for achieving good performance.

We experimentally searched for optimal values of batch size and learning rate to achieve high accuracy in image classification task.





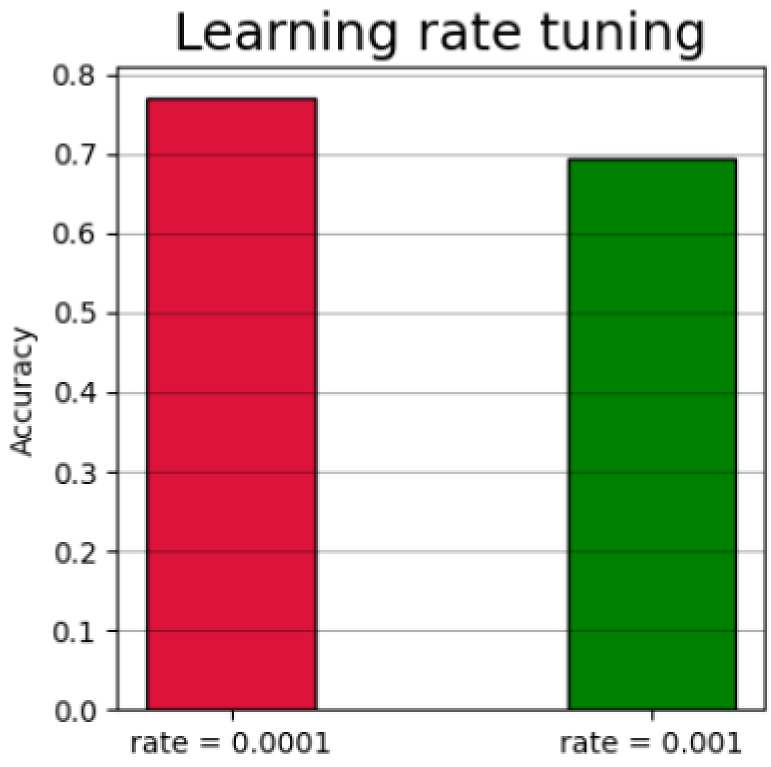
Learning rate of 0.0005 and 0.001 probably took too big steps. The optimal one was set to 0.0001.

The optimal batch size was the biggest one, which makes sense, as ImageNet contains 1000 object classes (dataset that alexnet was trained on). Therefore the model does need to see too many images in every single batch to perform well. We have decided to use batch size equal to 64.

MyCNN

Learning rate

In case of MyCNN model we decided to tune only learning rate as - due to complexity of the model - the tuning was very time-consuming.



With learning rate equal to 0.0001 the model had approximately 78% accuracy.

Experiments and results

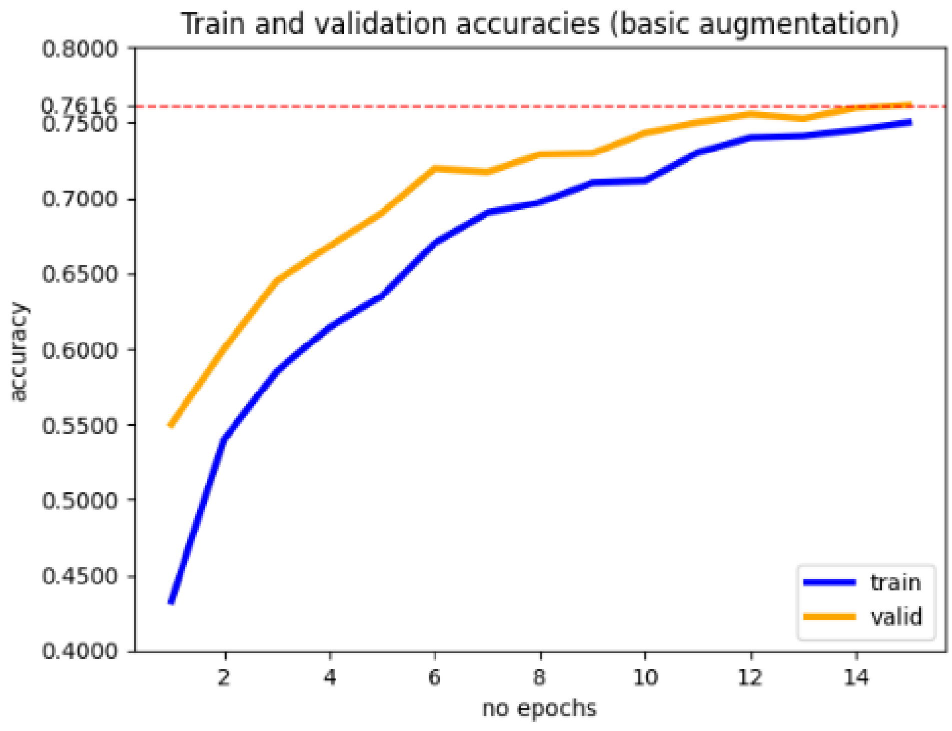
In this chapter we show the results of all conducted experiments. This includes:

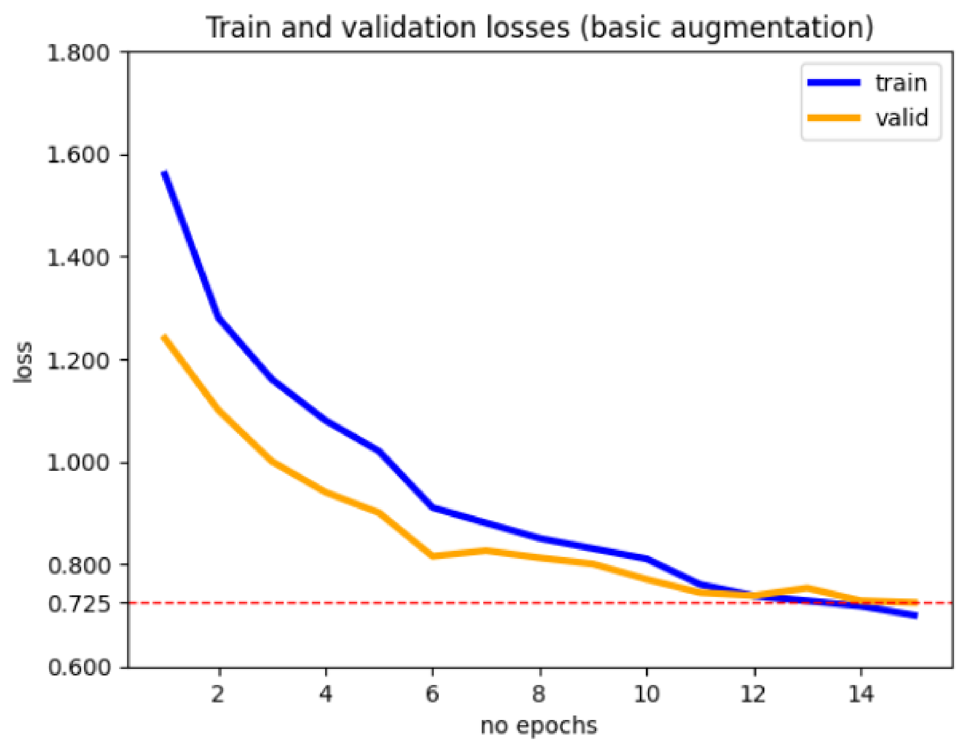
- Testing augmentation techniques and measuring their influence on accuracy of the models.
- Examining different types of ensembles and verifying their affect on the results.
- Showing confusion matrices for the models and investigating which classes are the models struggling with.

Augmentation

In this section we reveal results of data augmentation. Our strategy was to tune hyperparameters of networks on original dataset and then, with fixed parameters, test how augmentation influences accuracies and losses.

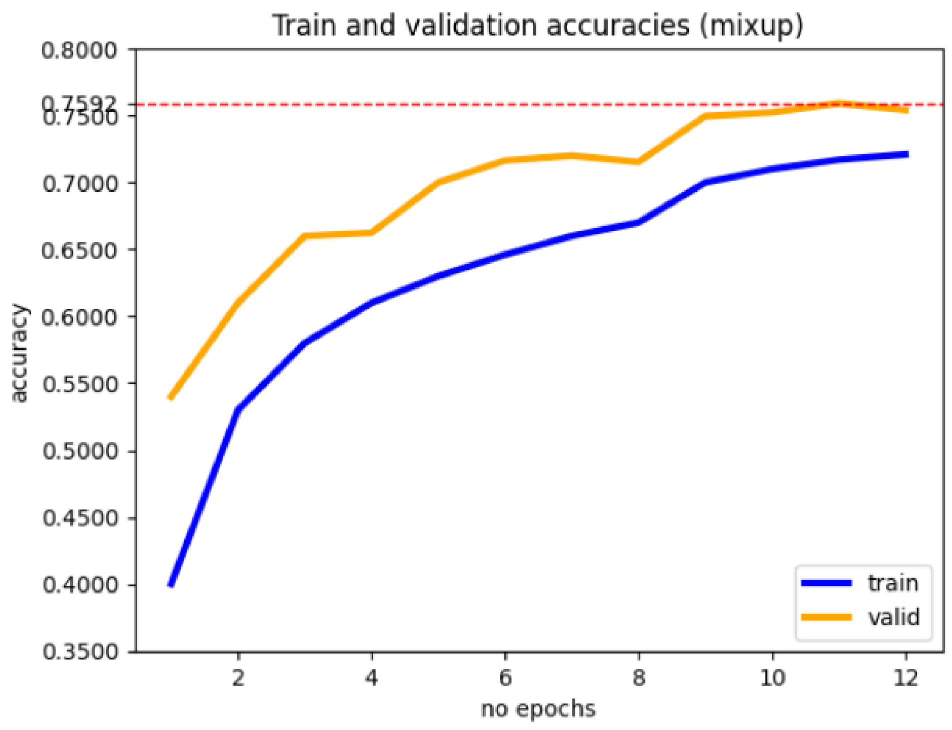
Basic augmentation (CNN_3_class model)

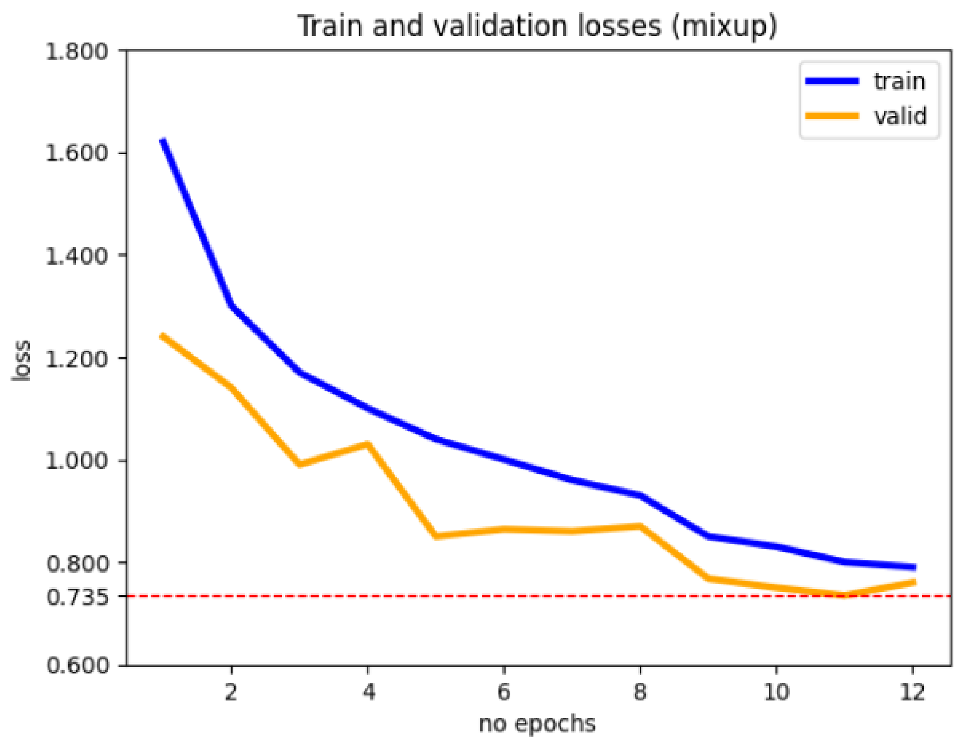




Basic augmentation technique brought big improvement for the model. As we recall, model on original data was performing best after 3 epochs and started to overfit later. With augmentation we do not experience overfitting even after 15 epochs. We also achieve the highest accuracy of the model - 76.16%.

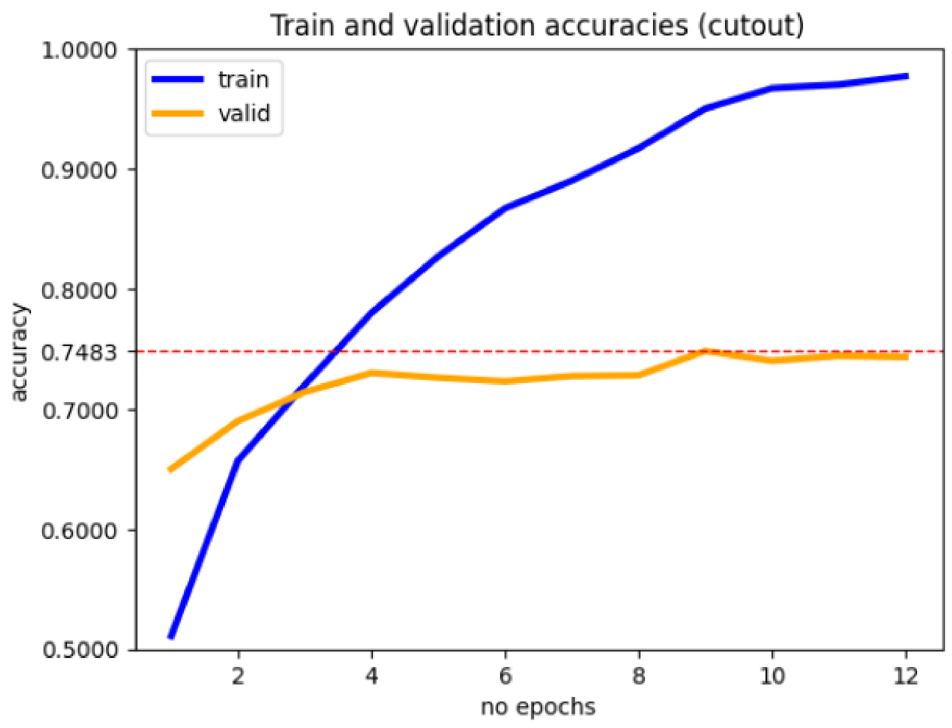
Mixup (CNN_3_class model)

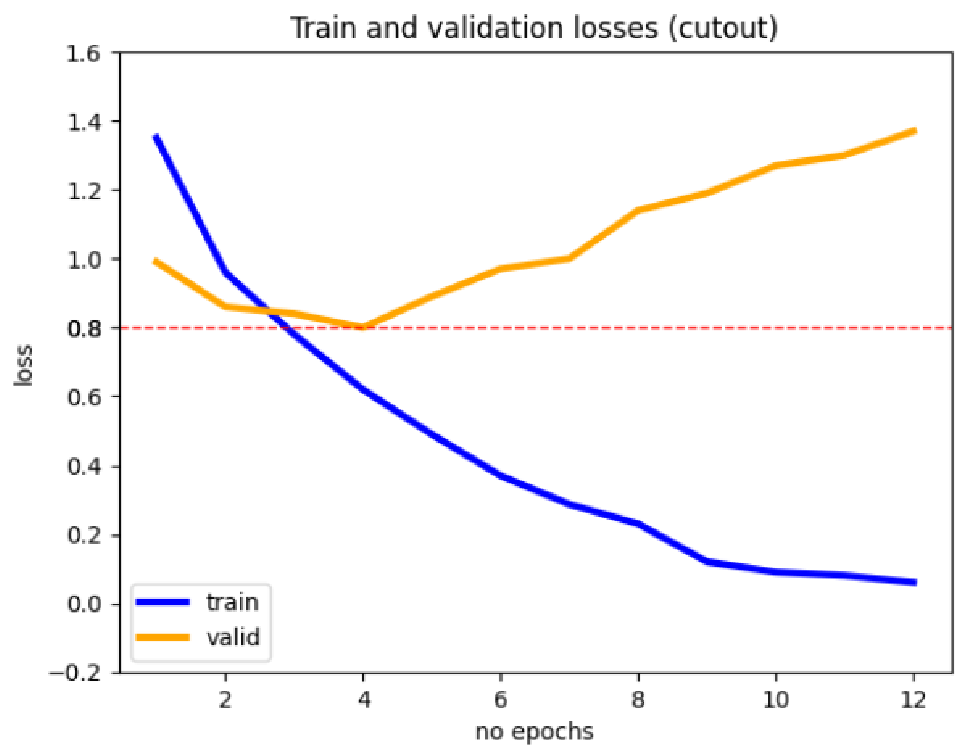




Mixup technique also gives big boost for the performance. Probably it is the most promising method out of all, as validation accuracy line is noticeably higher than the training one indicating, that we could train without overfitting for another couple of epochs. The validation loss also looks very appealing.

Cutout (CNN_3_class model)

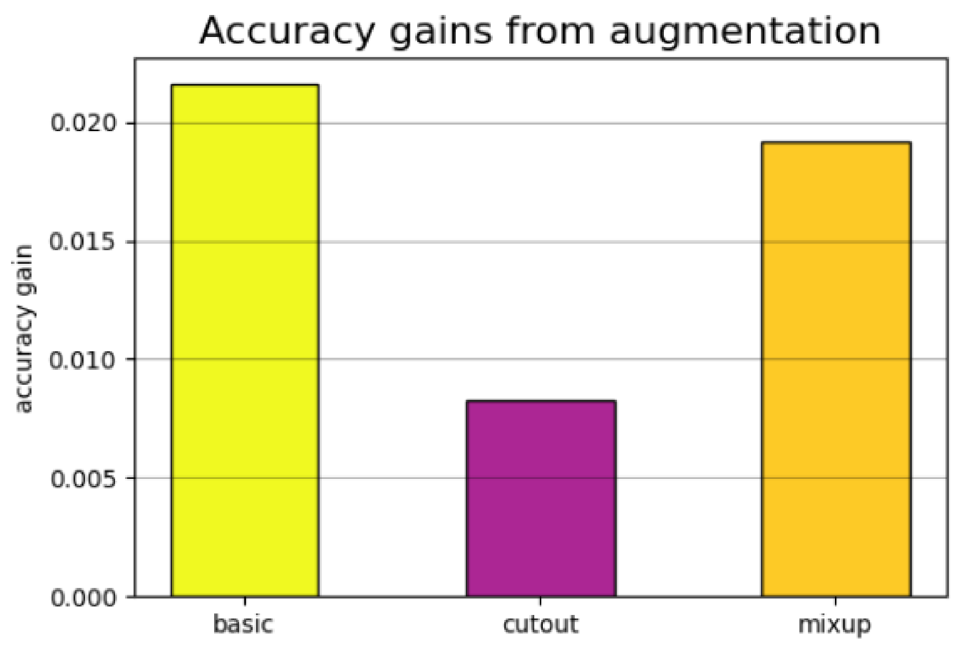




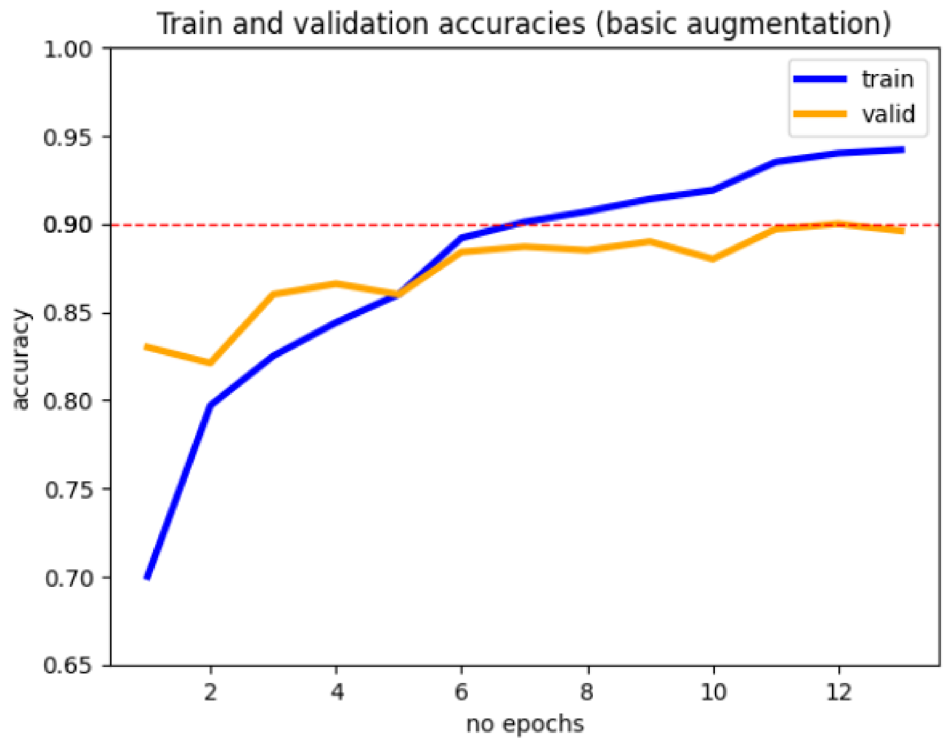
Cutout technique was the worst out of all, but still brought some improvement to the model. As we can see, the model started to overfit after around 5th epoch. The possible solution to this problem would be to either add dropout layer or add some other, basic augmentation along with cutout (as we did in case of Mixup augmentation).

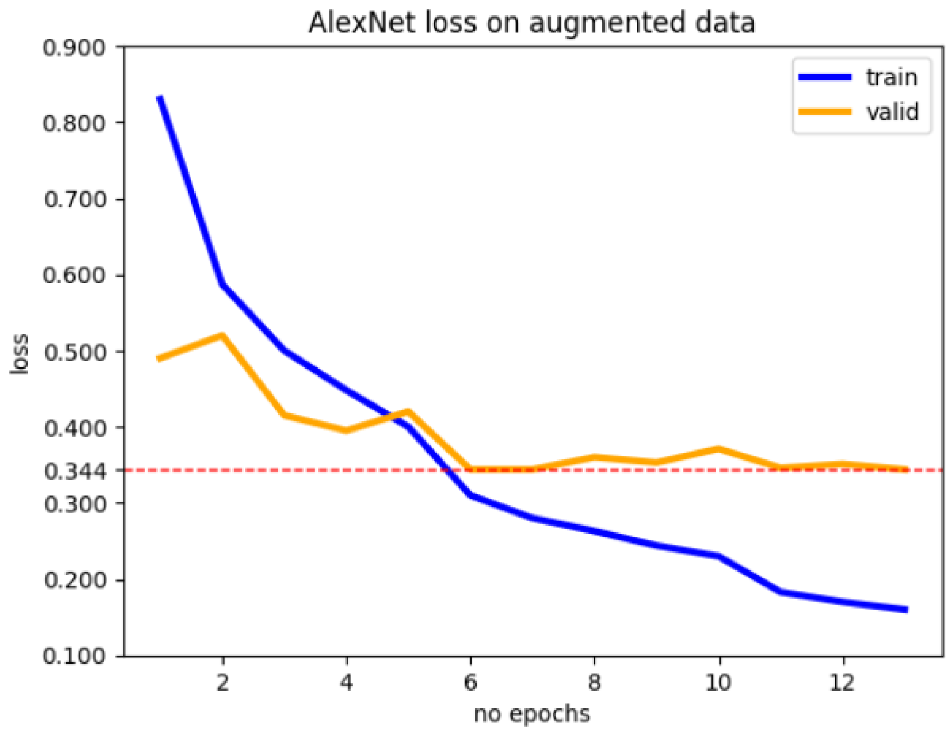
Summary

Below we summarize the results of data augmentation for CNN_3_class model.



Basic augmentation (PretrainedAlexNet)



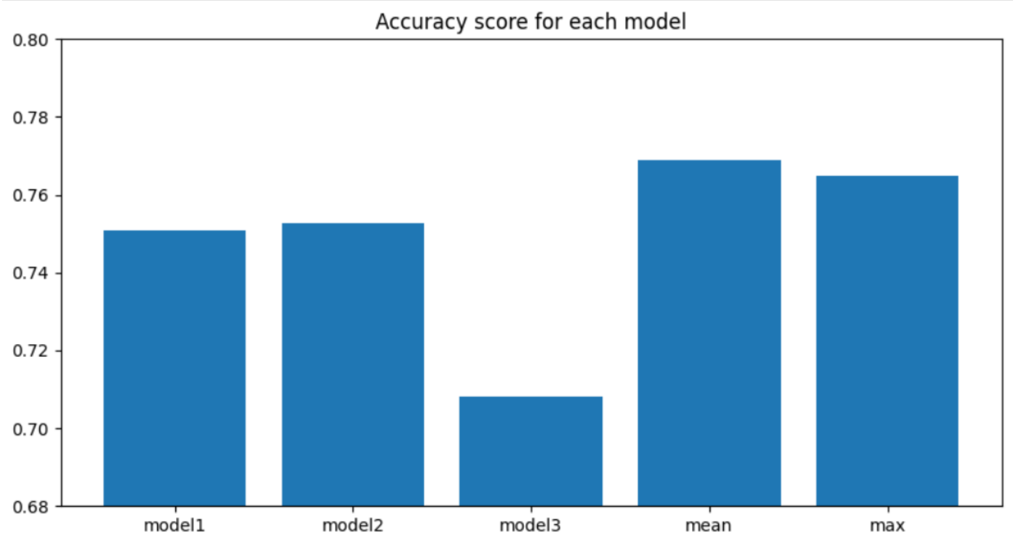


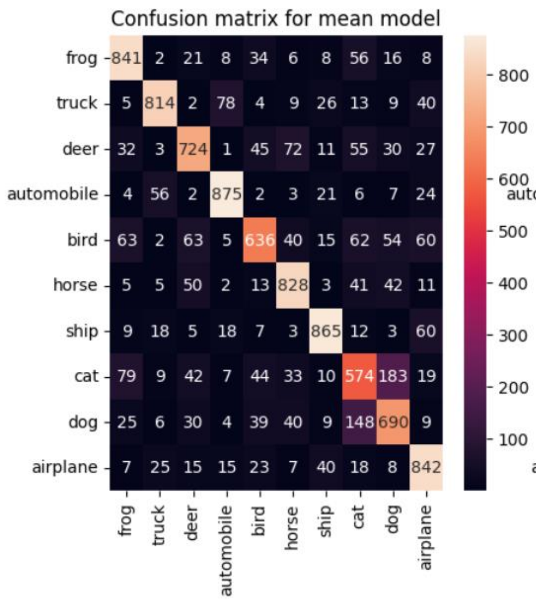
Results of training AlexNet on augmented data are solid, as we achieve 90% acccuracy after around 13 epochs. On the other hand, AlexNet trained on original data achieved over 90%, therefore in this case augmentation does not bring improvement

Ensemble

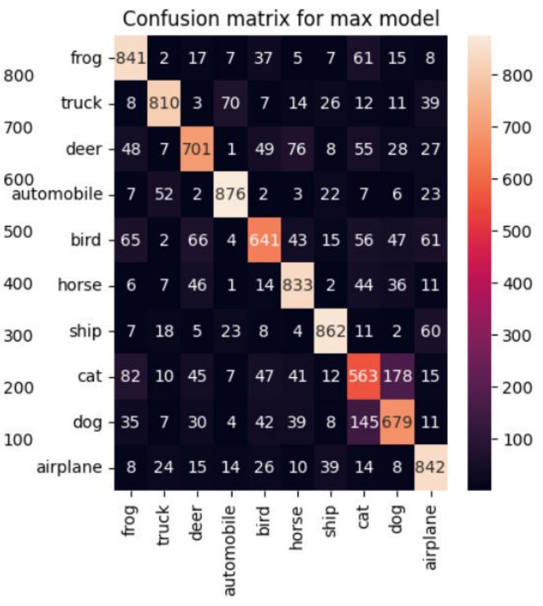
1. Compare ensemble model to the basic ones. Basic models are trained on original training set.

   To compare the accuracy of the individual models and the ensemble models, we will be using the accuracy metric and the confusion matrix. The accuracy metric measures the percentage of correctly classified observations. The confusion matrix is a table that shows the number of true positives, false positives, true negatives, and false negatives. It is useful for identifying the strengths and weaknesses of the models and for identifying which classes are being misclassified.

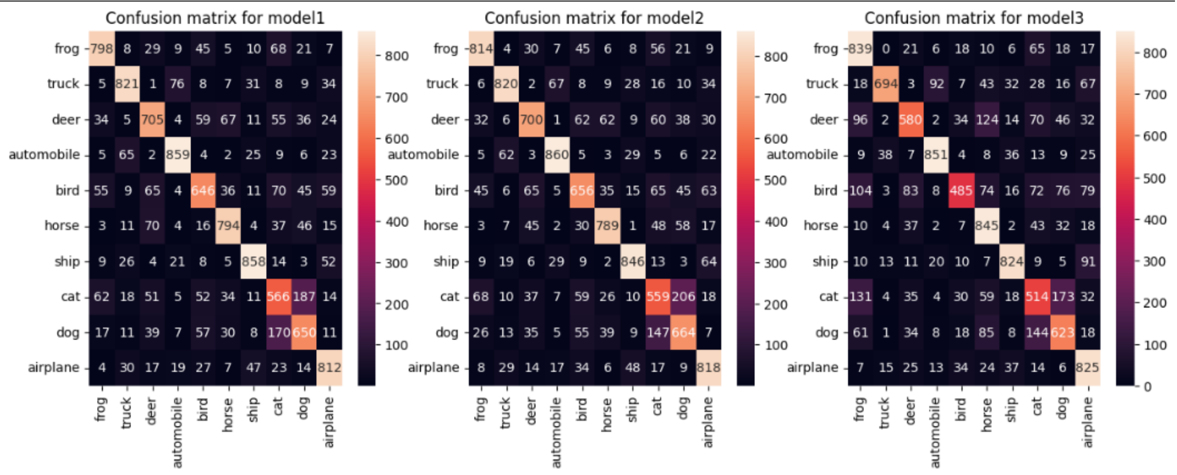




	frog	truck	deer	automobile	bird	horse	ship	cat	dog	airplane
frog	841	2	21	8	34	6	8	56	16	8
truck	5	814	2	78	4	9	26	13	9	40
deer	32	3	724	1	45	72	11	55	30	27
automobile	4	56	2	875	2	3	21	6	7	24
bird	63	2	63	5	636	40	15	62	54	60
horse	5	5	50	2	13	828	3	41	42	11
ship	9	18	5	18	7	3	865	12	3	60
cat	79	9	42	7	44	33	10	574	183	19
dog	25	6	30	4	39	40	9	148	690	9
airplane	7	25	15	15	23	7	40	18	8	842



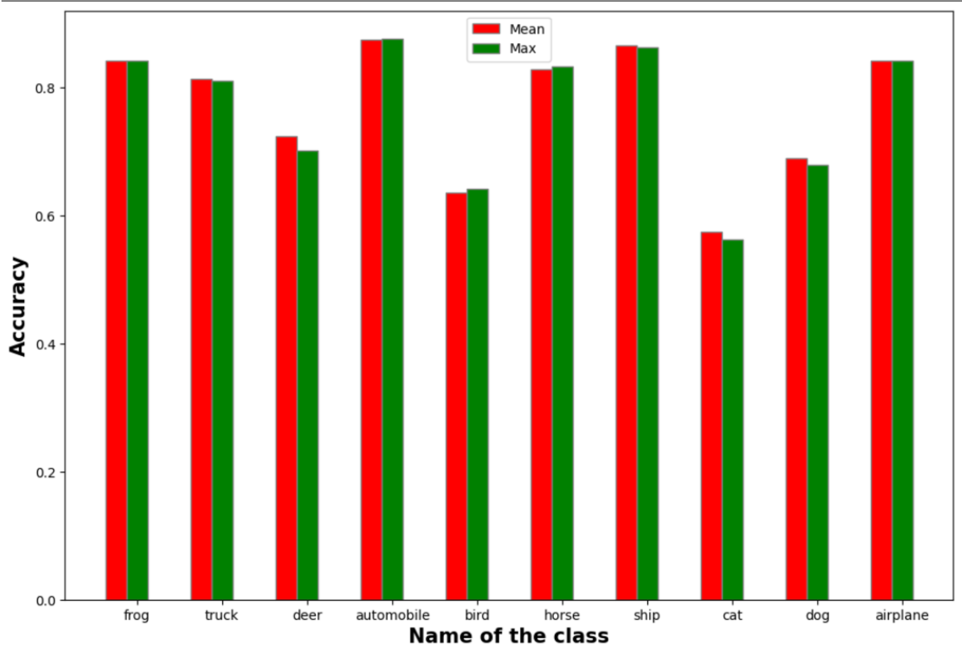
	frog	truck	deer	automobile	bird	horse	ship	cat	dog	airplane
frog	841	2	17	7	37	5	7	61	15	8
truck	8	810	3	70	7	14	26	12	11	39
deer	48	7	701	1	49	76	8	55	28	27
automobile	7	52	2	876	2	3	22	7	6	23
bird	65	2	66	4	641	43	15	56	47	61
horse	6	7	46	1	14	833	2	44	36	11
ship	7	18	5	23	8	4	862	11	2	60
cat	82	10	45	7	47	41	12	563	178	15
dog	35	7	30	4	42	39	8	145	679	11
airplane	8	24	15	14	26	10	39	14	8	842

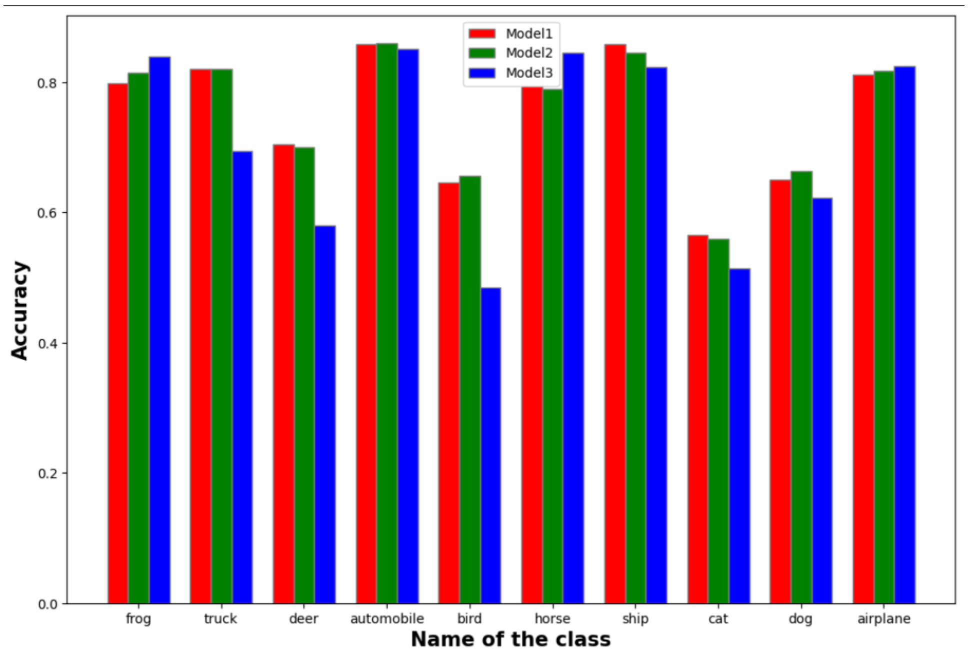


In conclusion, both ensemble methods - model_mean and model_max - perform better than the individual models. The advantage of ensemble methods is that they combine the strengths of individual models and reduce their weaknesses, resulting in improved accuracy and robustness of predictions.

Additionally, by examining the confusion matrix, we can observe that the models struggle with distinguishing between cats and dogs, which is an intuitive problem given the similarities between these animals.

That observation can be easily visualized on a bar plot.

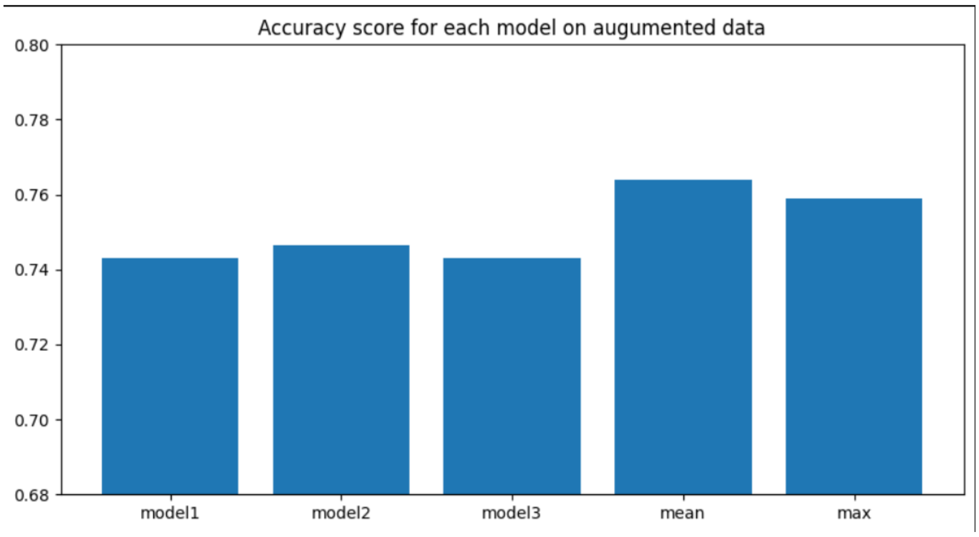




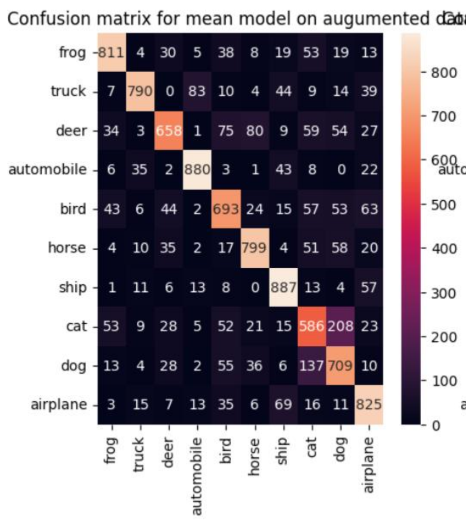
Combining accuracy metric and confusion matrix we receive a lot of information. Looking at the bar plot divided into class sections, we could say that cats and dogs labels can be mismatched with bird of deer with the same probability. However looking at the confusion matrix, despite having similar accuracy scores, we can see that cats and dogs are mismatched with each other while deer and birds are mismatched with all of the classes.

2. Compare ensemble model to the basic ones. Basic models are trained on the augmented training set.

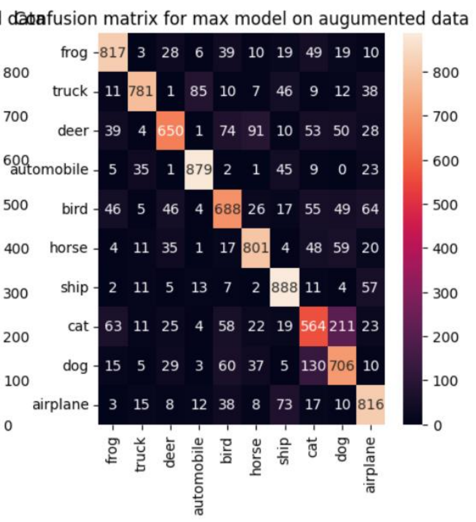
In this point we will again use accuracy and confusion matrix as a metrics to compare the models. Using the same method is good idea in that case, because we can separate other factors that could possibly have influence on the results.



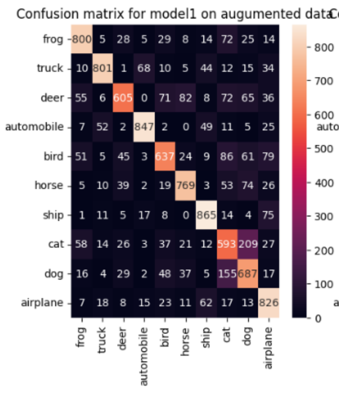
As we can see, models trained on the augmented data have mostly slightly lower accuracy than the models trained on the original data, however the ensemble models still perform better than the basic ones.



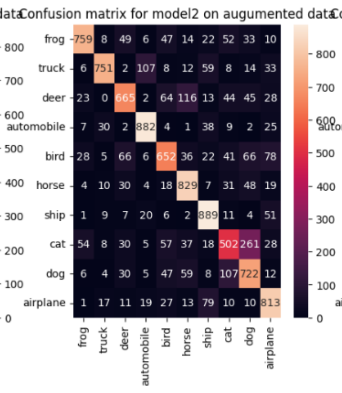
	frog	truck	deer	automobile	bird	horse	ship	cat	dog	airplane
frog	811	4	30	5	38	8	19	53	19	13
truck	7	790	0	83	10	4	44	9	14	39
deer	34	3	658	1	75	80	9	59	54	27
automobile	6	35	2	880	3	1	43	8	0	22
bird	43	6	44	2	693	24	15	57	53	63
horse	4	10	35	2	17	799	4	51	58	20
ship	1	11	6	13	8	0	887	13	4	57
cat	53	9	28	5	52	21	15	586	208	23
dog	13	4	28	2	55	36	6	137	709	10
airplane	3	15	7	13	35	6	69	16	11	825



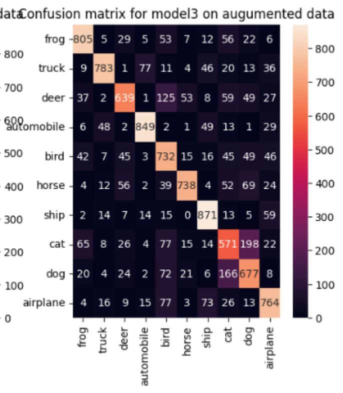
	frog	truck	deer	automobile	bird	horse	ship	cat	dog	airplane
frog	817	3	28	6	39	10	19	49	19	10
truck	11	781	1	85	10	7	46	9	12	38
deer	39	4	650	1	74	91	10	53	50	28
automobile	5	35	1	879	2	1	45	9	0	23
bird	46	5	46	4	688	26	17	55	49	64
horse	4	11	35	1	17	801	4	48	59	20
ship	2	11	5	13	7	2	888	11	4	57
cat	63	11	25	4	58	22	19	564	211	23
dog	15	5	29	3	60	37	5	130	706	10
airplane	3	15	8	12	38	8	73	17	10	816



	frog	truck	deer	automobile	bird	horse	ship	cat	dog	airplane
frog	800	5	28	5	29	8	14	72	25	14
truck	10	801	1	68	10	5	44	12	15	34
deer	55	6	605	0	71	82	8	72	65	36
automobile	7	52	2	847	2	0	49	11	5	25
bird	51	5	45	3	637	24	9	86	61	79
horse	5	10	39	2	19	769	3	53	74	26
ship	1	11	5	17	8	0	865	14	4	75
cat	58	14	26	3	37	21	12	593	209	27
dog	16	4	29	2	48	37	5	155	687	17
airplane	7	18	8	15	23	11	62	17	13	826

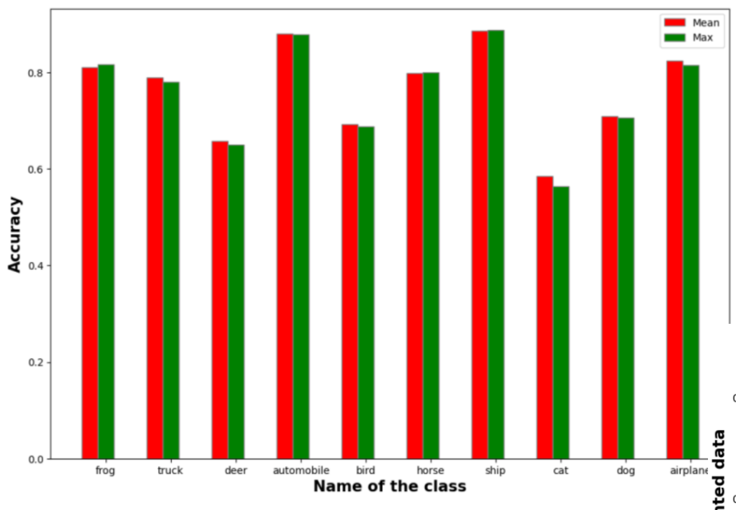


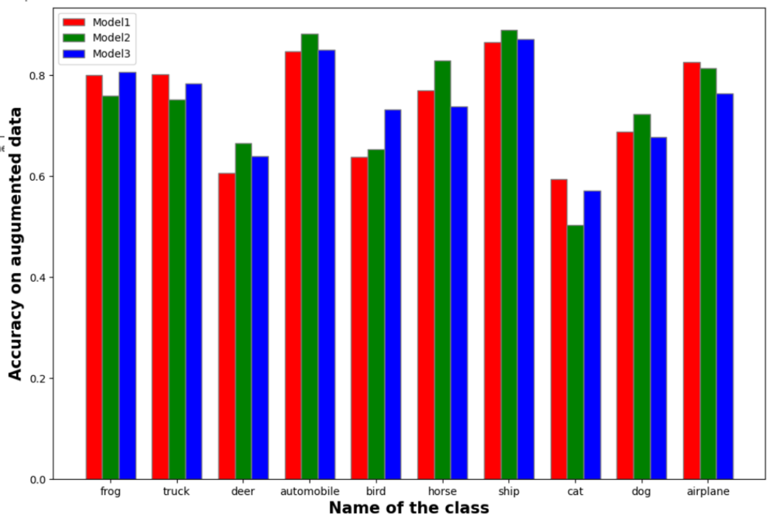
	frog	truck	deer	automobile	bird	horse	ship	cat	dog	airplane
frog	759	8	49	6	47	14	22	52	33	10
truck	6	751	2	107	8	12	59	8	14	33
deer	23	0	665	2	64	116	13	44	45	28
automobile	7	30	2	882	4	1	38	9	2	25
bird	28	5	66	6	652	36	22	41	66	78
horse	4	10	30	4	18	829	7	31	48	19
ship	1	9	7	20	6	2	889	11	4	51
cat	54	8	30	5	57	37	18	502	261	28
dog	6	4	30	5	47	59	8	107	722	12
airplane	1	17	11	19	27	13	79	10	10	813



	frog	truck	deer	automobile	bird	horse	ship	cat	dog	airplane
frog	805	5	29	5	53	7	12	56	22	6
truck	9	783	1	77	11	4	46	20	13	36
deer	37	2	639	1	125	53	8	59	49	27
automobile	6	48	2	849	2	1	49	13	1	29
bird	42	7	45	3	732	15	16	45	49	46
horse	4	12	56	2	39	738	4	52	69	24
ship	2	14	7	14	15	0	871	13	5	59
cat	65	8	26	4	77	15	14	571	198	22
dog	20	4	24	2	72	21	6	166	677	8
airplane	4	16	9	15	77	3	73	26	13	764

In that case we can still observe the problem with dog-cat classification.

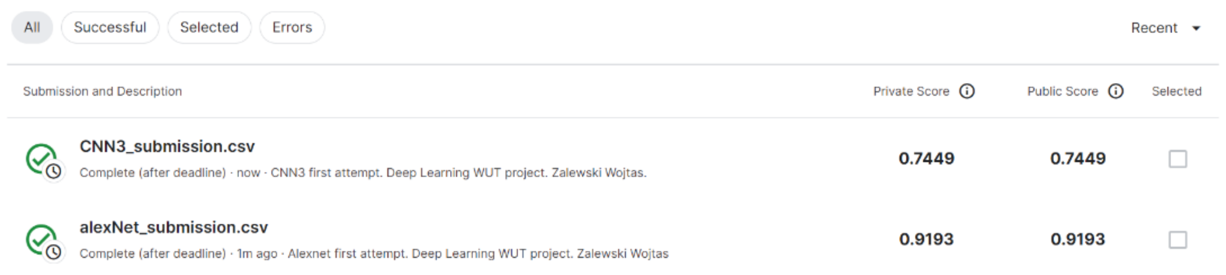




Summary on Tuning Multiple Neural Networks for Image Classification

In this part, we discuss our efforts to tune several neural networks for image classification tasks. Our aim was to identify the best-performing models and evaluate their performance on a new, previously unseen dataset.

After experimenting with several architectures and tuning various hyperparameters, we found that two models, CNN3 (on augmented data) and pre-trained AlexNet, performed the best on the training and validation sets. To assess their performance on an unknown dataset, we uploaded these models to Kaggle and tested them on a dataset of 300k images that were not part of our training or validation sets.



Submission and Description	Private Score	Public Score	Selected
CNN3_submission.csv Complete (after deadline) · now · CNN3 first attempt. Deep Learning WUT project. Zalewski Wojtas.	0.7449	0.7449	☐
alexNet_submission.csv Complete (after deadline) · 1m ago · Alexnet first attempt. Deep Learning WUT project. Zalewski Wojtas	0.9193	0.9193	☐

The results were promising, with CNN3 achieving an accuracy of almost 75%, which is a good result for a custom-built network. However, pre-trained AlexNet performed even better, achieving an accuracy of over 91%. This clearly demonstrates the potential of pre-trained models for image classification tasks.

In conclusion, our experiments show that pre-trained models such as AlexNet can achieve exceptional results in image classification tasks. It is clear that such models have great potential and should be used when the situation allows. Nevertheless, custom-built architectures such as CNN3 can still achieve good results and may be preferable in certain situations.

References:

1. “A survey on Image Data Augmentation for Deep Learning”, Connor Shorten & Taghi M. Khoshgoftaar, Journal of Big Data (link: https://journalofbigdata.springeropen.com/articles/10.1186/s40537-019-0197-0)
2. “Weighted Random Search for CNN Hyperparameter Optimization”, R. Andonie, A.C. Florea, International Journal of Computers Communications & Control, 04.2020 (link: https://arxiv.org/ftp/arxiv/papers/2003/2003.13300.pdf)
3. https://pyimagesearch.com/2021/07/19/pytorch-training-your-first-convolutional-neural-network-cnn/
4. https://medium.com/thecyphy/train-cnn-model-with-pytorch-21dafb918f48
5. https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/
6. https://adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/