16 April 2021

Image Classification with Convolutional Neural Networks

Deep Learning Project no. 2

Paulina Pacyna, 290600
Mateusz Wójcik, 290639
Mateusz Szysz, 298911

1 Introduction

The goal of this report is to present our solution to the problem of CIFAR images classification with convolutional neural networks. There are two main sections. The first one is focused on describing our custom architecture of the network. Whereas the other one concerns presenting performance of some widely-used pre-trained models and popular architectures used in practice.

While the full code is included in the attached files, this article points out only the most important facts like overview of the design, performance of different models and interpretations of the results.

2 Custom Architectures

In this section we describe our custom model created without using any pre-trained solution or any popular named architecture. In fact we prepared many models with small differences in the architecture and chose 13 best of them. Due to that fact we could also investigate the power of ensembling.

2.1 Architecture details

One of the models have been created with the code shown below.

```
model = Sequential()

model.add(Conv2D(filters=32, kernel_size=(3,3),input_shape=(32, 32, 3),
                 activation='relu', padding = 'same'))
model.add(Conv2D(filters=32, kernel_size=(3,3),input_shape=(32, 32, 3),
                 activation='relu', padding = 'same'))

model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.3))

model.add(Conv2D(filters=64, kernel_size=(3,3),input_shape=(32, 32, 3),
                 activation='relu', padding = 'same'))
model.add(Conv2D(filters=64, kernel_size=(3,3),input_shape=(32, 32, 3),
                 activation='relu', padding = 'same'))

model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.3))

model.add(Conv2D(filters=128, kernel_size=(3,3),input_shape=(32, 32, 3),
                 activation='relu', padding = 'same'))
model.add(Conv2D(filters=128, kernel_size=(3,3),input_shape=(32, 32, 3),
                 activation='relu', padding = 'same'))

model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.3))

model.add(BatchNormalization())

model.add(Flatten())

model.add(Dense(256, activation='relu', kernel_regularizer=l2(0.001)))
model.add(Dropout(0.4))

model.add(Dense(10, activation='softmax'))
```

Listing 1: Custom model

There are three sets of two convolutional layers. All of them use 3x3 kernels, but they differ in the number of filters. We decided to choose the most popular approach in which the number of filters grows with deeper layers. The idea behind this method is that initially the network has to learn the most simple objects like lines or edges. Then it should learn more complex ones which require more filters. We used default parameter in case of the stride (which is 1x1), because the images did not have many pixels, so there was no need to reduce it significantly in this way. Also in each layer of the network we settled on ReLu as the activation function, because it's the most popular approach due to the lack of the vanishing gradient problem.

After each set of convolutional layers, there are pooling layers with 2x2 kernels. They are useful for reducing number of parameters without loosing much knowledge. In our

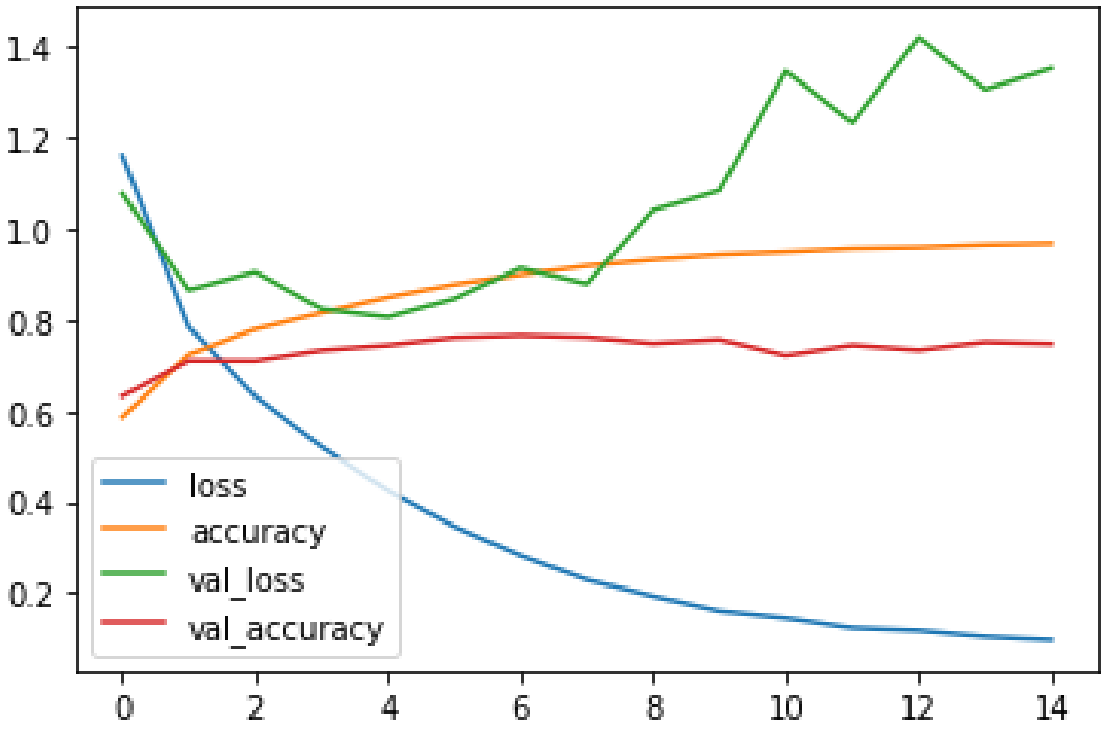


Figure 1: Loss function and accuracy of model without dropouts and regularization.

models, there was used max pooling instead of average pooling, because we wanted to extract sharp elements better instead of smoothing the images.

After convolutional layers we added batch normalization which might result in reducing number of epochs needed for learning the most accurate parameters.

To avoid the problem of overfitting, we added dropouts. In this case the probability of dropping neurons in given iteration of backward and forward pass was 0.3, but in other models we tried different values. On the one hand it makes the model simpler and on the other one it makes the network doesn't rely on small number of neurons but has to learn general features of given objects. What is more, in the last hidden layer we added L2 regularization which discourages learning too complex and flexible network. The importance of preventing overfitting is presented in the chart below.

As can be seen the value of the loss function on the training set converges to 0. Whereas the loss on the validation set is growing very sharply. It's caused by the fact that the network without dropouts learns from some random relationships between pixels on given images.

2.2 Learning parameters

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

early_stop = EarlyStopping(monitor='val_loss',patience=20)


model.fit(X_train, y_cat_train, epochs=200,
        validation_data=(X_test, to_categorical(y_test,10)),
        callbacks=[early_stop])
```

Listing 2: Custom model

For the optimizer we used Adam method which is currently the most popular and effective one. As a number of epochs we chose 200. However, to decrease number of unnecessary steps we added early stopping. Due to this fact, our model stopped training after 89 epochs and did not have to make additional 111 steps. From efficiency point of view it was very important because the model had to learn more than $8,000,000$ parameters which took quite a lot of time even with this optimization method.

Another solution that we added to make the model learns faster was scaling all the pixel values to the range of $[0; 1]$ which is a standard procedure in images classification.

2.3 Datasets used in learning

The main set used for learning was kaggle train dataset which consists of 50.000 observations. As a validation set we used original CIFAR dataset with $10,000$ observations.

The other one was very useful to set early stopping and to tune parameters, but also to visualize performance which is impossible in case of kaggle test set because the labels haven't been provided.

2.4 Learning procedure

The chart below shows how the values of the loss function and the accuracy for both training and validation datasets have been changing across epochs. As may be seen, initially the values of the loss function are very similar on both sets. However at some point the loss function on validation set stops decreasing whereas on the training set it's still smaller and smaller. Probably if the early stoppping wasn't applied, all the consecutive steps of the learning would be completely unnecessary.

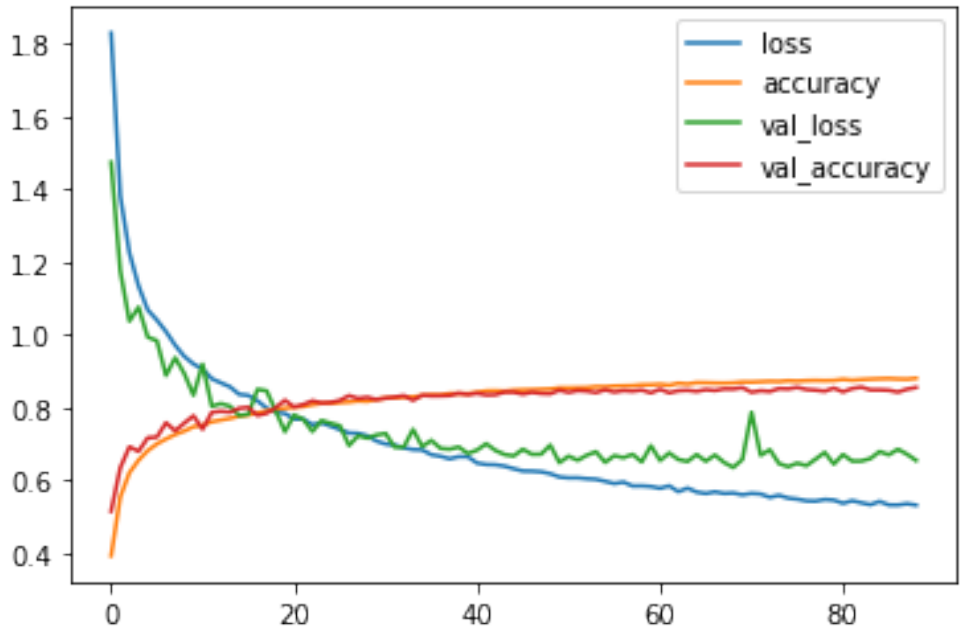


Figure 2: Loss function and accuracy values depending on epoch number.

2.5 Results

Finally, we can present the detailed results on the validation dataset and values of accuracy reported on kaggle.

	precision	recall	f1-score	support
0	0.85	0.88	0.87	1000
1	0.91	0.94	0.93	1000
2	0.86	0.75	0.80	1000
3	0.75	0.71	0.73	1000
4	0.82	0.86	0.84	1000
5	0.80	0.77	0.79	1000
6	0.91	0.89	0.90	1000
7	0.86	0.91	0.88	1000
8	0.89	0.93	0.91	1000
9	0.88	0.91	0.90	1000
accuracy			0.85	10000
macro avg	0.85	0.85	0.85	10000
weighted avg	0.85	0.85	0.85	10000

The data above shows the performance of the model divided into different classes. As may be seen the model predicts the most accurately images from class number 1 which are automobiles. Whereas the worst performance may be reported on the class number 3 which are cats. However it's worth to notice that all the performance values are greater than 0.7 while random classification would give performance around 0.1.

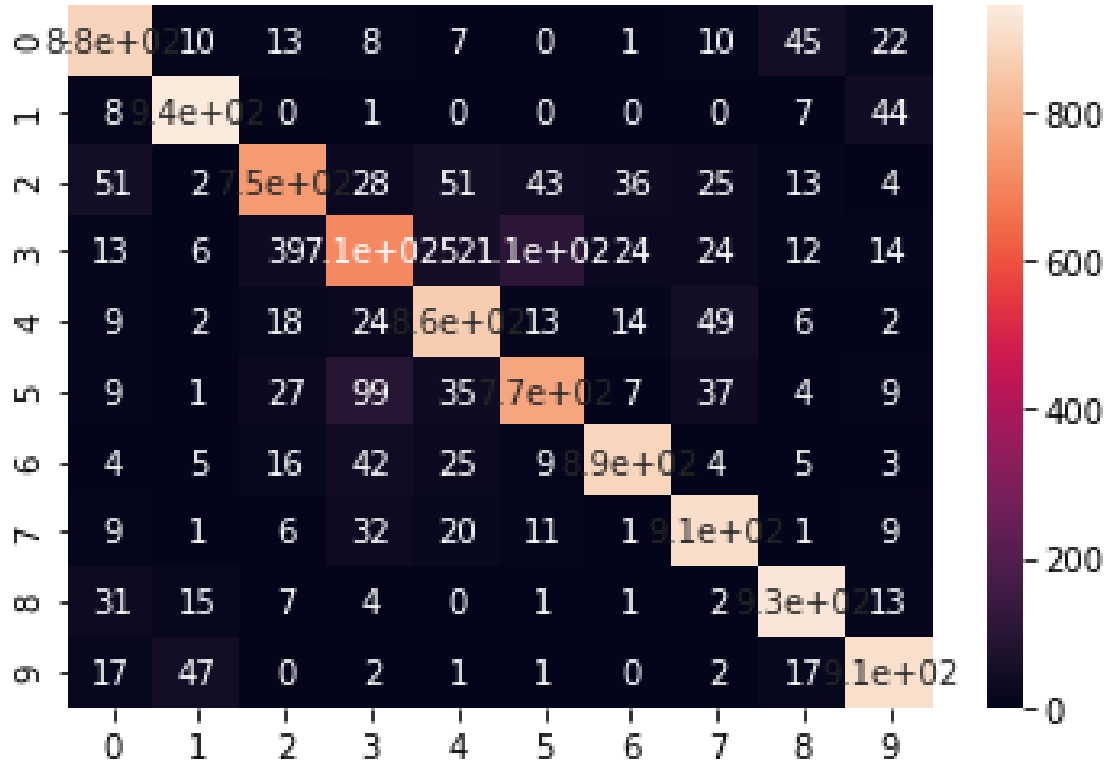


Figure 3: Confusion matrix heatmap.

Similar conclusion may be drawn from analyzing the heatmap of the confusion matrix. Majority of values lie on the diagonal which shows correct predictions. However there is clearly seen that the model has difficulties with classifying cats.

At the end, we checked the performance of our model on kaggle. The result we got was equal to 0.8549 which was quite surprising for us, because we expected a little worse results without using pre-trained models or some complex architectures.

However, as we mentioned previously we created 13 different models. On each of them the reported accuracy on kaggle was in range $0.835 - 0.86$, but when we used all of them for better predictions we got the following results.

Model	Accuracy
Custom Model	0.8549
Hard Voting	0.8848
Soft Voting	0.8887
Weighted Soft Voting	0.8902

As anticipated, combining a few models with moderate accuracy gives substantially better predictions than each of the models separately. It's due to the fact that models

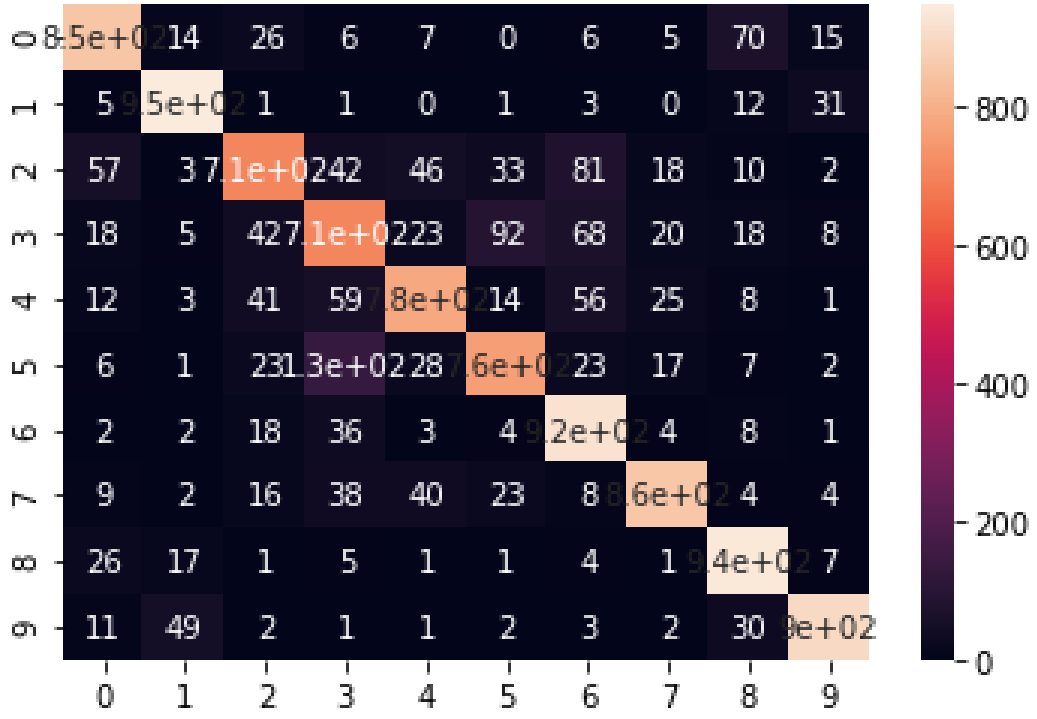


Figure 4: Confusion matrix heatmap for another model.

differently misclassify some images. For example one of our other models, whose confusion matrix heatmap is shown below, predicts cats a little better.

Averaging these different models led to the evident increase in the performance. The final accuracy equals 0.8902.

Soft voting seems to be better choice than the usual majority voting because it takes into account the degree of certainty and not only the rough prediction.

Apart from that, adding weights for different models may increase the accuracy in the soft voting, because some models may be recognised to be more reliable than the others.

2.6 Summary of the Custom Model

As we mentioned at the beginning, we did not want to describe all of the remaining models separately because they differ only in values of some parameters and give quite similar accuracy. All the details may be checked in the Jupyter notebook files.

During searching for the best architecture we evaluated many different models. Creating more simple architecture gave substantially worse accuracy, while more complex

ones took much more time for training but did not give clearly better results. However there are some well-established architectures and pre-trained models whose parameters were tuned very carefully in many experiments independent from this specific problem of CIFAR dataset. We will introduce examples of them in the subsequent sections of the report.

3 Pre-trained Models and Existing Architectures

One of our targets was to utilize available pre-trained models. We have chosen ResNet pre-trained on 'imagenet' and DenseNet-201 model for this purpose. We have also tested existing architecture

3.1 ResNet

ResNet is relatively new architecture. It became popular after winning ILSVC challenge in 2015. ResNet introduced a new "identity shortcut connection" concept, which helped to tackle the vanishing gradient problem appearing in deep architectures. An "identity shortcut connection" is a direct connection that skips one or more layers.

We extended the ResNet50 architecture by adding two up sampling layers at the top and 4 blocks consisting of batch normalization, dense layer and dropout. This model will be called as 'base model'. In this model we also use data augmentation and a learning rate scheduler. The data augmentation possibilities are rotation, zooming in and horizontal flip. The scheduler divides the learning rate by 100 when there is no change in validation accuracy for 3 epochs.

We will test 4 modifications to this model:
- 'freezing' the ResNet layer weights, which means that they will not be adjusted during the learning procedure. This reduces the execution time significantly, however it may have an impact on accuracy.
- no learning rate scheduler (constant learning rate)
- setting a custom scheduler with following rule: Let's denote initial learning rate as lr and epoch number as n:

$$\begin{cases} lr & n \leq 5 \\ lr/10 & 5 \leq n \leq 20 \\ lr/100 & 20 \leq n \end{cases}$$

- no data augmentation

The plot 5 shows the accuracy in each epoch for each model. We added an 'Early stopping' callback which stops the training if there is no change in validation loss. Therefore, the lines have different lengths.

As we see, the model with custom scheduler outperforms other models in terms of high accuracy and quick convergence. Therefore, we will use this model for ensemble training.

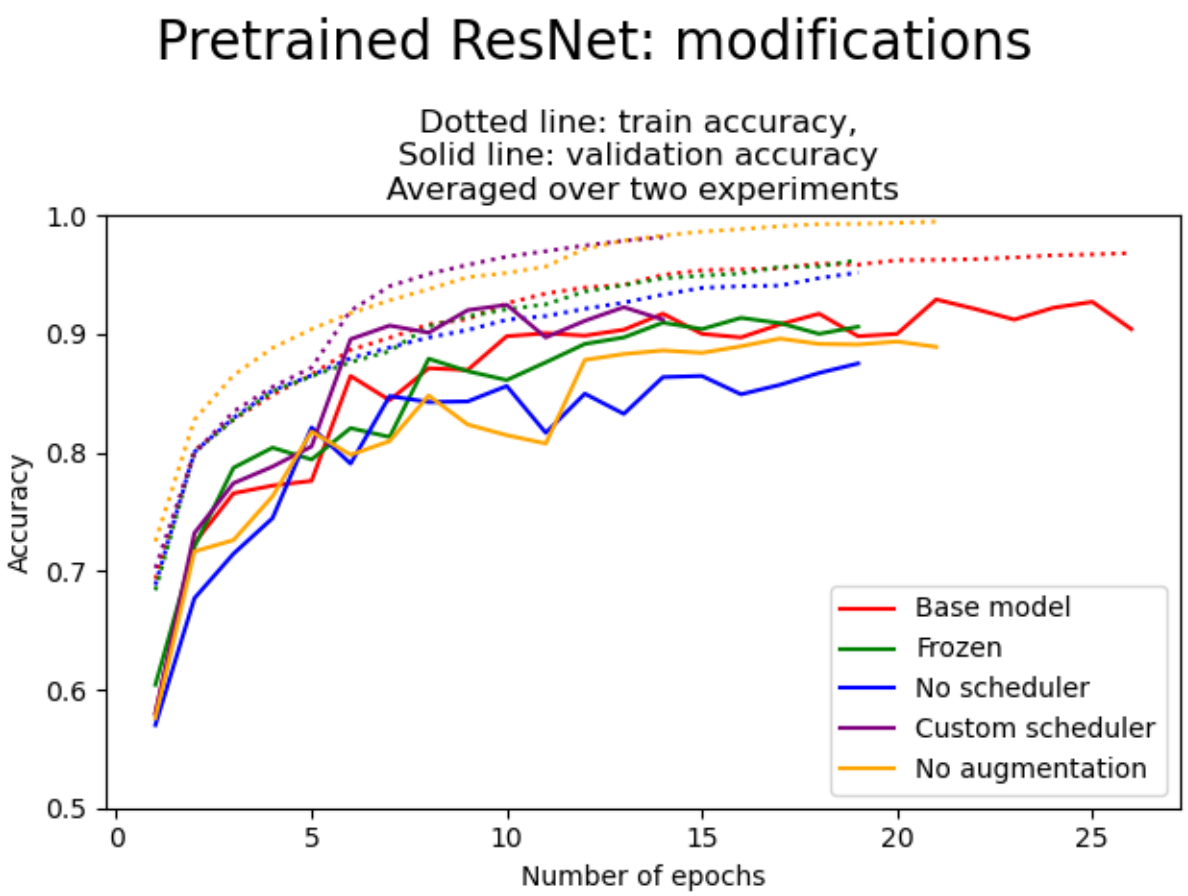


Figure 5: Accuracy for different models

3.2 DenseNet-201

DenseNet-201 is a CNN consisting of 201 layers. It was pretrained on more than a million images from the ImageNet database. Images can be classified into 1000 object categories, e.g. keyboard, mouse, pencil, cat, dog, car, etc. The default image input size is 224×224. Therefore, we first needed to resize images from the training data for the purpose of the project. To adjust the pre-trained network, we used transfer learning which is a technique that applies to deep neural networks that have been extensively trained with large-sized datasets.

In more details, we decided to *freeze* base model, downloaded from Keras module, with additional top infrastructure consisting of Dense layers trained with the CIFAR10 dataset. We have added one dense layer of 256 units with *elu* activation, dropout layer, and dense layer of 10 units with *softmax* activation as an output layer. We used Adam compiler and categorical cross-entropy loss function. To control learning rate, we tested two approaches: reduction of learning rate when a validation accuracy metric has stopped improving and exponential decay. Also early stop and checkpoint for each epoch was provided.

We tested different batch sizes for training. Data augmentation was also applied (rotation, width and height shift, zoom, filling), but the results appeared to be less satisfactory with comparison to plain data or data with horizontal flip.

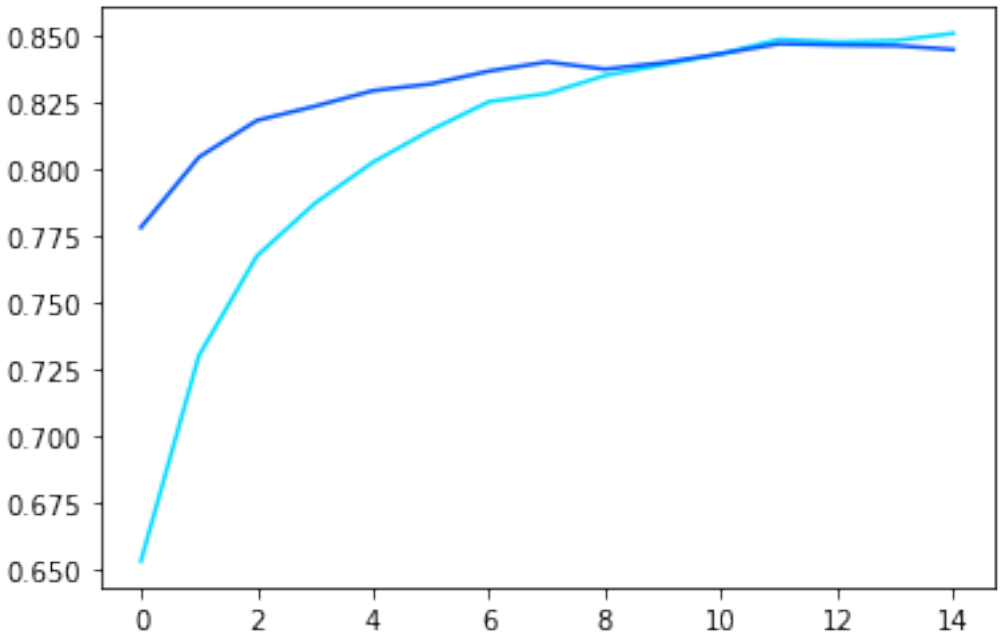


Figure 6: Training and Validation Accuracy vs Number of Epochs

3.3 ConvPool-CNN

We have also provided an existing architecture for learning from scratch. The architecture is based on the article *Striving for Simplicity: The All Convolutional Net*. The model features a common pattern where several convolutional layers are followed by a pooling layer. On the other hand, the final layers consists of a global average pooling layer instead of several fully-connected layers.

In particular, *relu* activation and *softmax* activation were used. We conducted up to 120 epochs for the training with the batch size from the set $\{16, 32, 64, 128\}$. Adam optimizer and categorical cross-entropy loss function were applied. We used a constant learning rate in this case. The training data was generated with data augmentation: horizontal flip, width and height shift, fill, or rotation.

This model obtained quite good results with about 0.88 accuracy in a Kaggle competition.

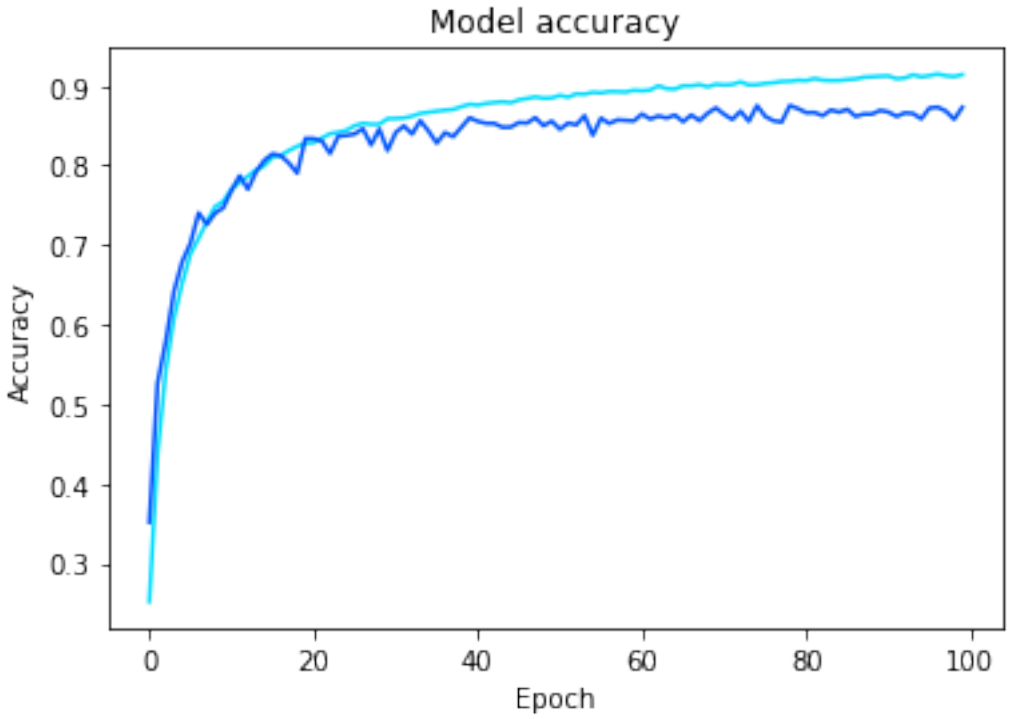


Figure 7: Training (deep sky blue) and Validation (blue) Accuracy

4 Final results

Finally, we could combine all the results using ensemble learning. The results of our experiments are shown in the table below.

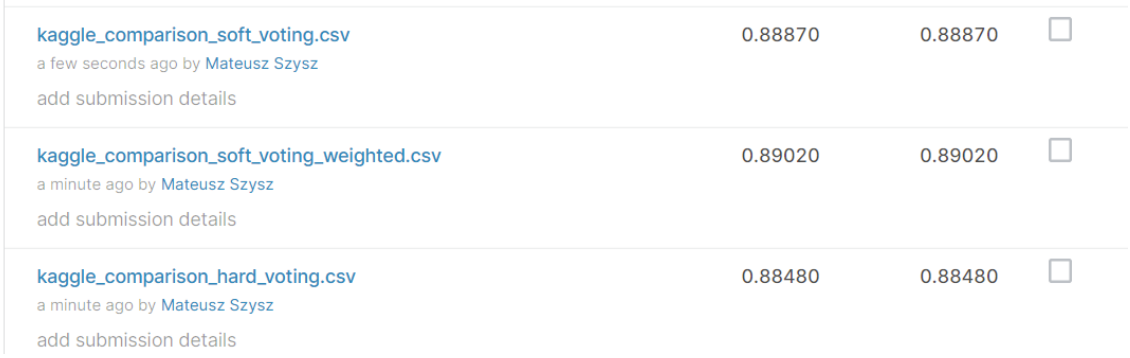
Model	Accuracy
ResNet	0.9
DenseNet	0.7739
ConvPool	0.8812
Custom WSV	0.8902
Soft Voting	0.9186
Weighted Soft Voting	0.9205

As can be seen, we got the best accuracy on the ResNet. However values gained from the ConvPool and values from weighted soft voting in custom models are quite similar. Combining these models gave even better accuracy which is finally equal to 0.9205.

5 Sources

- lecture materials
- www.kaggle.com/c/cifar-10/ (data)
- www.cs.toronto.edu/ kriz/cifar.html (dataset description)
- machinelearningmastery.com/how-to-develop-a-cnn-from-scratch-for-cifar-10-photo-classification/ (architecture proposition)
- www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques/ (regularization methods)
- www.tensorflow.org/api_docs/python/tf/keras (tensorflow.keras documentation)
- https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035
- https://medium.com/@andrew.dabydeen/transfer-learning-using-resnet50-and-cifar-10-6242ed4b4245 (inspiration for ResNet architecture)
- https://medium.com/@pierre.beaujuge/classifying-images-from-the-cifar10-dataset-with-pre-trained-cnns-using-transfer-learning-9348f6d878a8
- https://www.mathworks.com/help/deeplearning/ref/densenet201.html
- https://towardsdatascience.com/ensembling-convnets-using-keras-237d429157eb

6 Attachments



kaggle_final_new_13.csv 30 minutes ago by Mateusz Szysz add submission details	0.84200	0.84200	☐
kaggle_final_new_12.csv 31 minutes ago by Mateusz Szysz add submission details	0.83150	0.83150	☐
kaggle_final_new_11.csv 31 minutes ago by Mateusz Szysz add submission details	0.84700	0.84700	☐
kaggle_final_new_10.csv 32 minutes ago by Mateusz Szysz add submission details	0.83650	0.83650	☐
kaggle_final_new_10.csv 32 minutes ago by Mateusz Szysz add submission details	0.83650	0.83650	☐
kaggle_final_new_9.csv 33 minutes ago by Mateusz Szysz add submission details	0.83920	0.83920	☐
kaggle_final_new_8.csv 33 minutes ago by Mateusz Szysz	0.85230	0.85230	☐
kaggle_final_new_7.csv 34 minutes ago by Mateusz Szysz add submission details	0.84180	0.84180	☐
kaggle_final_new_6.csv 34 minutes ago by Mateusz Szysz add submission details	0.83900	0.83900	☐
kaggle_final_new_5.csv 34 minutes ago by Mateusz Szysz add submission details	0.85090	0.85090	☐
kaggle_final_new_4.csv 35 minutes ago by Mateusz Szysz add submission details	0.83940	0.83940	☐
kaggle_final_new_3.csv 36 minutes ago by Mateusz Szysz add submission details	0.84320	0.84320	☐
kaggle_final_new_2.csv 36 minutes ago by Mateusz Szysz add submission details	0.85000	0.85000	☐
kaggle_final_new_1.csv 37 minutes ago by Mateusz Szysz	0.85490	0.85490	☐

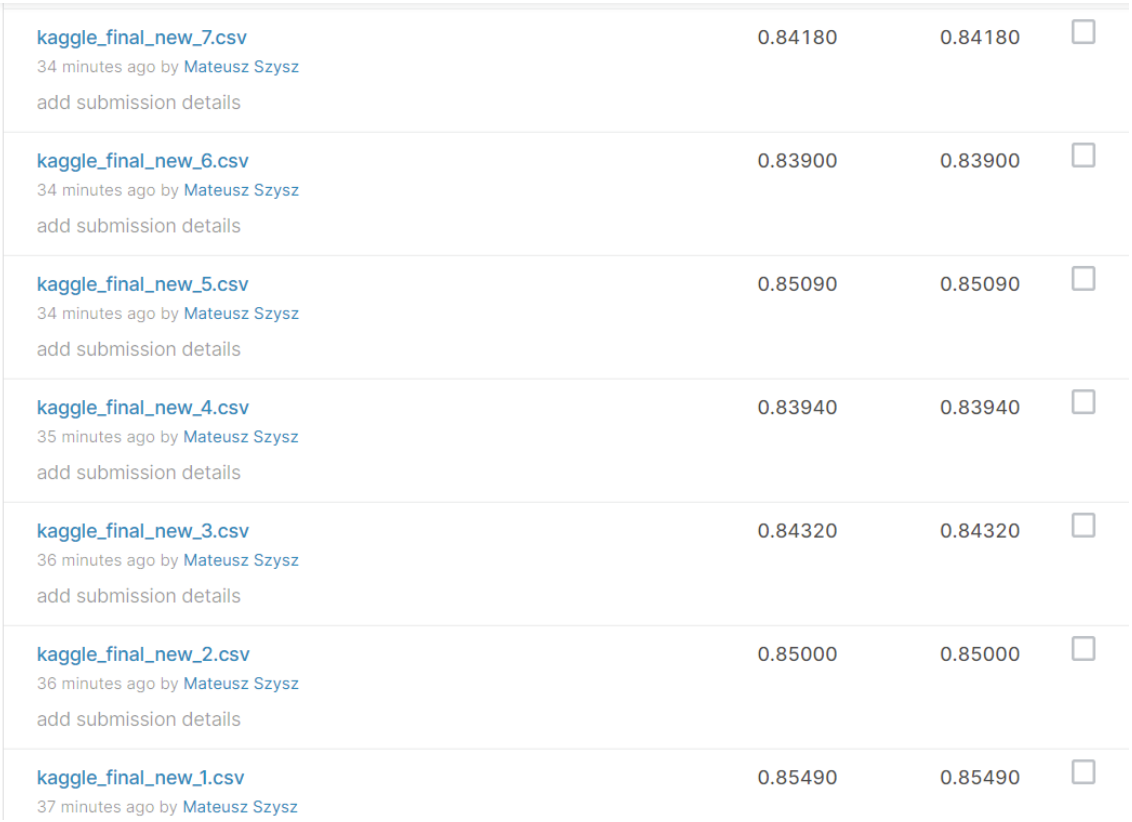


Figure 8: Kaggle results on custom models

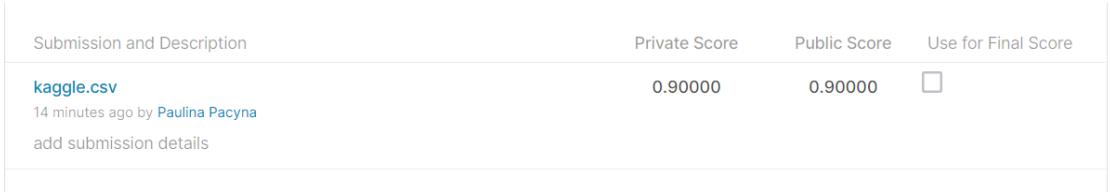


Figure 9: Kaggle results on the pretrained ResNet model (with custom scheduler)

Figure 10: Kaggle results on the Conv-Pool-CNN model

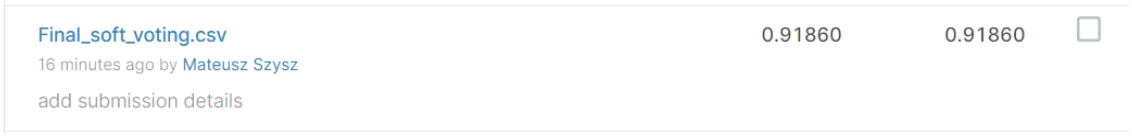


Figure 11: Kaggle final soft voting results

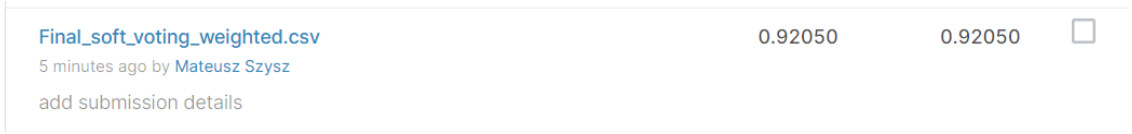


Figure 12: Kaggle final weighted soft voting results