Chapter 7 Results and Evaluation

Due to the computational requirements of the Convolutional Neural Network architectures in training time, the models were trained on dedicated software on the Google Cloud Platform. The virtual machine instance, has the following specifications:

CPU	**Intel Xeon 2.9 GHz**
GPU	NVIDIA TESLA K80 24GB
RAM	8GB
Persistent storage	30GB
OS Image	Intel® optimized Deep Learning Image: TensorFlow 1.13.1 m23 (with Intel® MKL-DNN/MKL and CUDA 10.0)

Notice that both the CPU and RAM memory are with standard specifications and similar hardware can be found in the majority of personal computers and laptops nowadays. However, both of the components are used only for the fetching and preprocessing of the dataset and the actual training of the neural network is executed mainly on the GPU. The CPU and RAM do not affect performance in a direct manner, however they can be used for the data augmentation and preprocessing if executed before training. In the experiments conducted in this project all data augmentations are done in batches in train time and therefore the component that contributes most for the performance and train time is the GPU – NVIDIA TESLA K80. An important characteristic of the GPU is the on-chip memory, as the more memory there is on the CPU, the larger batch size can be used in training and the faster the model converges. Finally, on the virtual machine was preinstalled a specialized Ubuntu OS image. This was necessary, because without the CUDA libraries, the TensorFlow deep learning framework cannot use the GPU effectively and this will affect performance negatively. The scripts for all models were written as Jupyter Notebooks to allow for fast prototyping and adjusting network hyperparameters. TensorBoard, the visualization tool of TensorFlow was used for visualizing the training metrics (train and validation accuracy and loss) during the fitting of the framework. Details for the configuration of TensorBoard can be found in Appendix A.

The experiments conducted throughout the project can be divided in four categories:

- No data augmentation
- Standard data augmentations
- Histogram equalization
- Constraining weights in the interval [0,1]

For each category, each of the investigated Convolutional Neural network architectures were trained and evaluated (namely VGG16, VGG19, ResNet50 and Xception). The model performance was evaluated over the Cifar10 benchmark dataset, containing 50 000 train images (image size of 32x32 pixels), belonging to one of 10 classes. In order to speed up training and convergence time, transfer learning was applied and the models were initially loaded with pretrained weights – the parameters extracted from the corresponding architecture trained on the Imagenet dataset. Each of the model's tops (the top part of the architecture, responsible for the classification of the features extracted by the Convolutional layers) was replaced with a custom top that consists of GlobalAveragePooling2D layer, a Fully-Connected layer with 1024 neurons and finally the Fully-Connected layer with softmax activation function provided the output class probabilities. As described earlier, this technique reduces significantly the number of parameters in the VGG architectures and thus speeds up both training and inference time. The optimizer used for the training initially was RMSProp with the default learning rate 1e-3. However, the models were not learning at all and the accuracy was constantly fluctuating around 0.1. The difference between training and validation loss (see Chapter 6 Validation and Verification for more information on the train data split) was too big with training accuracy reaching values of up to 98%, while both validation and test accuracy staying at a maximum of 86%. This indicated that the networks were overfitting on the training data. To reduce the overfitting, the optimizer was changed to Stochastic Gradient Descent (SGD) with momentum of 0.9. Furthermore, the learning rate was increased to 1e-2. The learning rate proved to be too big and the optimizer was taking too big steps in the gradient direction, which was fixed by reducing the rate to 1e-4 after which the networks started learning. The loss used for the training was categorical cross-entropy which works well with the optimizer and the multiclass

dimension of the problem. As the whole dataset is too big to fit on memory both in terms of RAM and On-GPU memory, the training was conducted in batches. The size of those batches (or batch size) is another important parameter and usually bigger size lead to faster convergence as there is more and more diverse data for each epoch, which allows for more accurate estimation of the gradient. The batch size was set to 512, as this was the largest value fitting into the TESLA K80 GPU memory, however if a GPU with more memory is available, the batch size can be increased accordingly.

7.1. No data augmentation

In the experiments without data augmentation, the data was just normalized by subtracting the mean and dividing by the standard deviation. Then as described earlier, the top layers of the original architectures were removed and replaced with custom layers – GlobalAveragePooling and two Fully-Connected layers.

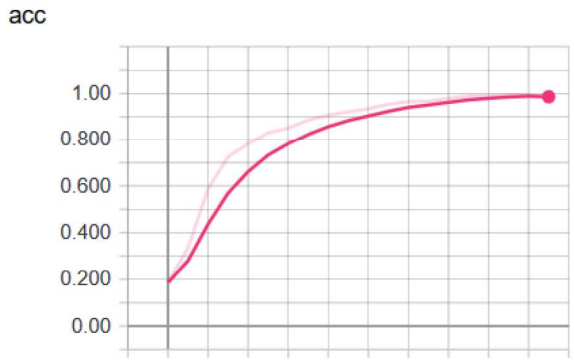
```
x = GlobalAvgPool2D()(x)
x = Dense(1024, activation='relu')(x)
x = Dense(classes, activation='softmax')(x)
```

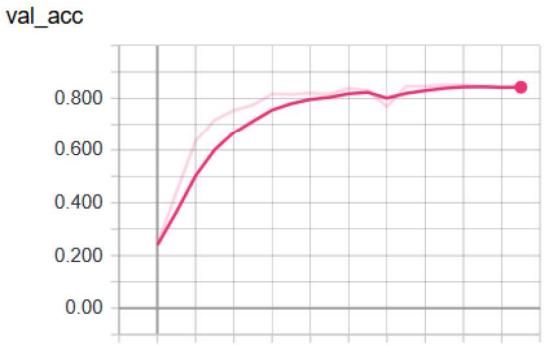
All models showed fast convergence and good accuracy result, however overfitting was still present in them, least noticeable in VGG16 as it is the smallest architecture and needs less data to train in general. To further reduce the overfitting, stronger regularization can be implemented, however that involves changing the models on per layer level, while similar regularization results were achieved with data augmentations in the following sections.

7.1.1. VGG16

Graph/20190323-171150

Batch size	512
Epochs	20
Validation split	0.2
Learning rate	1e-2
Momentum (Stochastic gradient descent)	0.9
Test accuracy	0.8372999995231628
Test loss	0.832587801361084
Training time	12:21

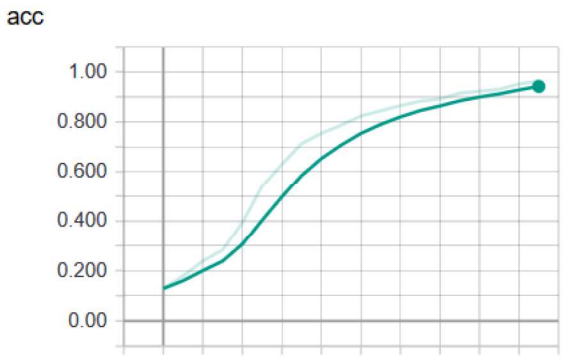


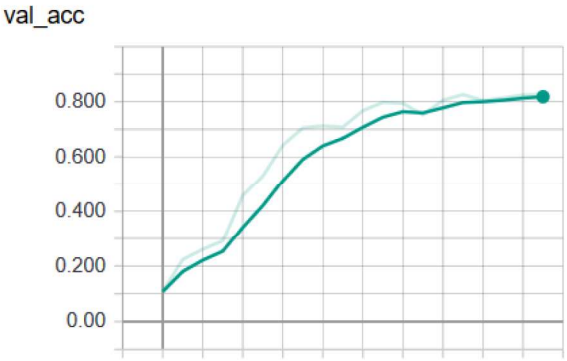


7.1.2. VGG19

Graph/20190323-173412

Batch size	512
Epochs	20
Validation split	0.2
Learning rate	1e-2
Momentum (Stochastic gradient descent)	0.9
Test accuracy	0.8206999995231629
Test loss	0.7503448402404785
Training time	11:21



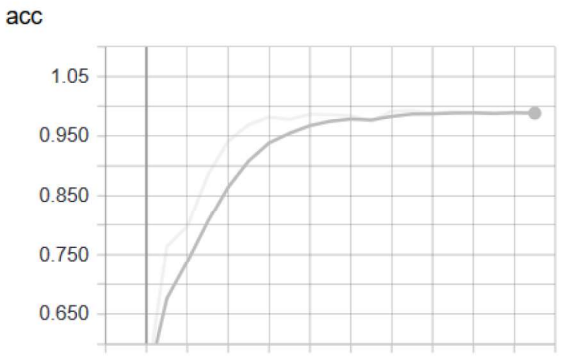


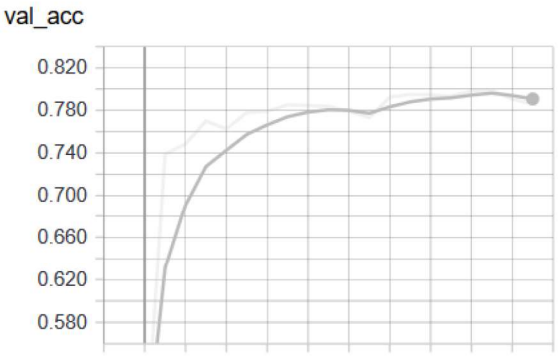
7.1.3. ResNet50

Graph/20190323-175405

Batch size	512
Epochs	20
Validation split	0.2
Learning rate	1e-2
Momentum (Stochastic gradient descent)	0.9

Test accuracy	0.7673
Test loss	1.0069275924682617
Training time	12:19

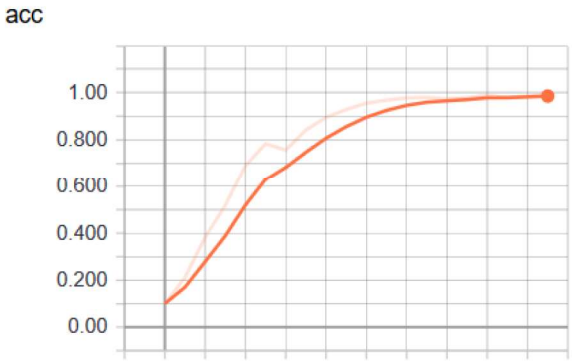


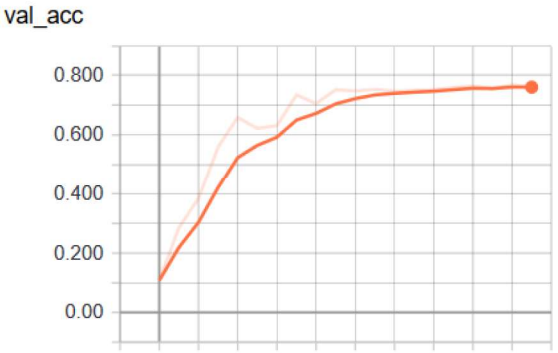


7.1.4. Xception

Graph/20190323-181556

Batch size	512
Epochs	20
Validation split	0.2
Learning rate	1e-2
Momentum (Stochastic gradient descent)	0.9
Test accuracy	0.7544
Test loss	1.342096117591858
Training time	12:02





7.2. With standard data augmentation

In this set of experiments standard data augmentations were applied to the training dataset. Initially, image rotations of 45 and 90 degrees were applied, however that had detrimental effect on the training as both train and test accuracy dropped. A possible reason for that is that most of the images both in the train and test datasets are relatively straight. This shows that data augmentations should be adjusted to the specific dataset and problem. For example, a dataset with facial shots will not need rotational augmentations, while one of ships in the sea might benefit greatly from that type of augmentation as satellites can take aerial shots from different angles. The final set of augmentations applied is the following:

Width_shift_range	0.2
Height_shift_range	0.2
Zoom_range=.2	0.2
Horizontal_flip	True
Vertical_flip	True

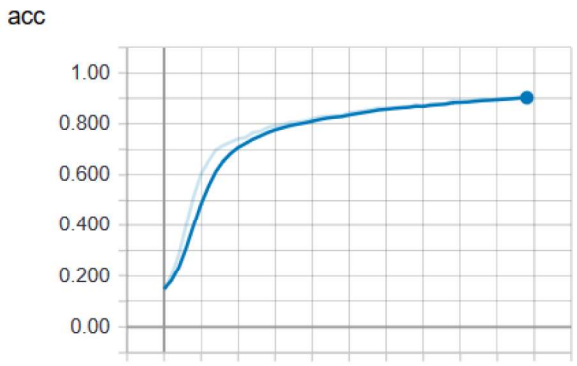
Width_shift and height_shift represents translations of the original image in the horizontal and vertical direction respectively and the range is a percentage of the shift as part of the image width (or hight). Similarly, zoom_range closes up the image. Horizontal_flip and vertical_flip flip the image in the correspondent direction. The augmentations are applied randomly and an example augmented image can be shifted horizontally by 10%, zoomed 20% and flipped vertically. As expected, applying standard data augmentations had a regularizing effect on the networks and the training and validation accuracy had significantly closer values and similar curves. However, applying image augmentations slows down the training process and to achieve similar results the number of epochs had to be increased from 20 to 50. Comparing the training time for VGG16, it increased from 12:21 to 28:02 minutes.

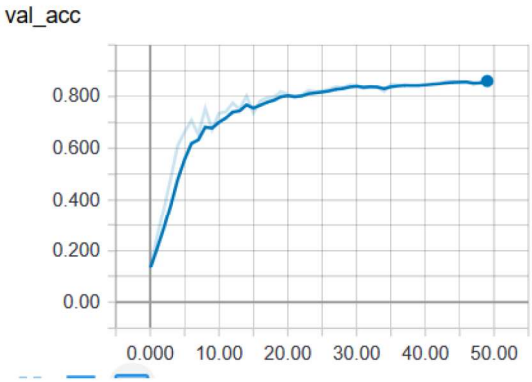
7.2.1. VGG16

Graph/20190323-083309

Batch size	512
Epochs	50
Validation split	0.2
Learning rate	1e-2
Momentum (Stochastic gradient descent)	0.9

Test accuracy	0.8604
Test loss	0.4318272342205048
Training time	28:02

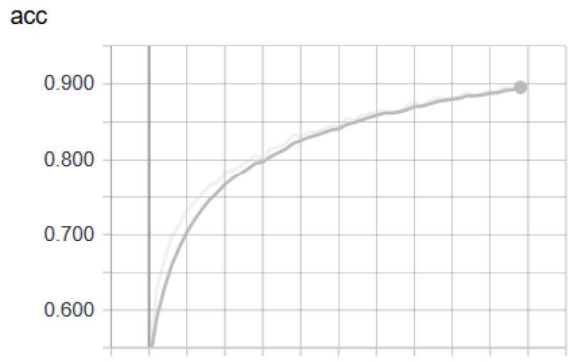


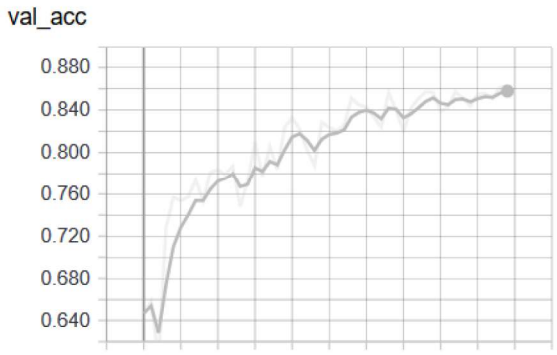


7.2.2. VGG19

Graph/20190323-091602

Batch size	512
Epochs	50
Validation split	0.2
Learning rate	1e-2
Momentum (Stochastic gradient descent)	0.9
Test accuracy	0.8503
Test loss	0.4876495380282402
Training time	22:40

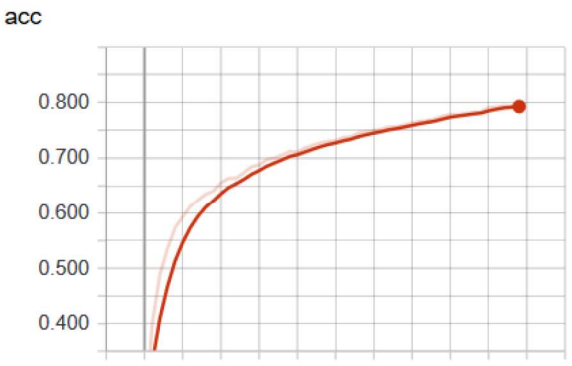


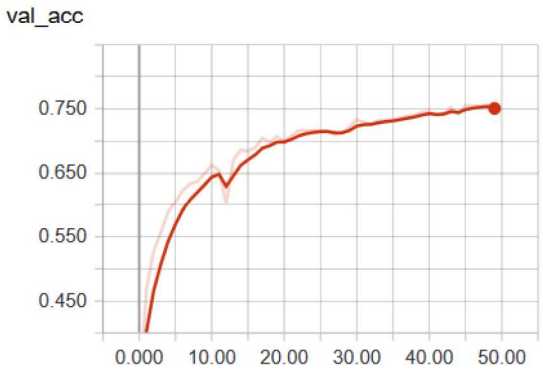


7.2.3. ResNet50

Graph/20190323-114440

Batch size	512
Epochs	50
Validation split	0.2
Learning rate	1e-3
Momentum (Stochastic gradient descent)	0.9
Test accuracy	0.7377
Test loss	0.7925023315429688
Training time	34:03

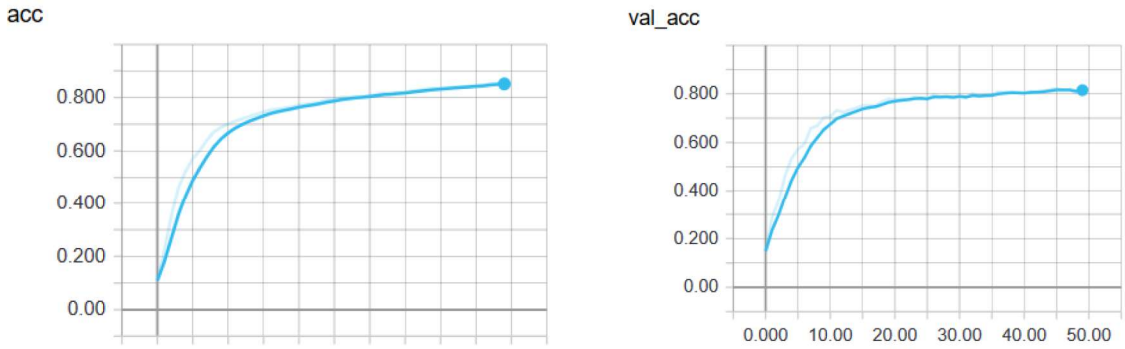




7.2.4. Xception

Graph/20190323-122422

Batch size	512
Epochs	50
Validation split	0.2
Learning rate	1e-2
Momentum (Stochastic gradient descent)	0.9
Test accuracy	0.8096
Test loss	0.5691010305643082
Training time	32:53



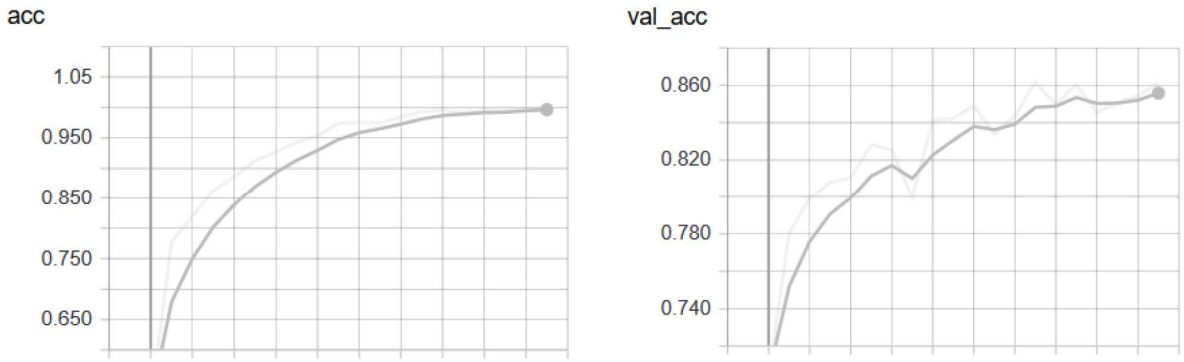
7.3. Histogram equalization

As described in section 5.1.2., histogram equalization techniques can help bring up important features in the input images, which could be missed otherwise as part of the background noise. After experimenting with the three types – contrast stretching, plain histogram equalization and adaptive histogram equalization, it was decided the dataset to be preprocessed with contrast stretching. Plain histogram equalization produces too unpredictable results, perhaps due to the fact that it can overamplify noise in images (see section 5.1.2.2.). Adaptive histogram equalization theoretically mitigates that issue, however practically it slowed down the training process drastically. If a faster implementation of the algorithm can be developed, then it will probably produce results comparatively better than the other two types. In the conducted experiments there were applied no other data augmentation to show the effect of histogram equalization as a standalone augmentation.

7.3.1. VGG16

Graph/20190323-131155 time 09:36

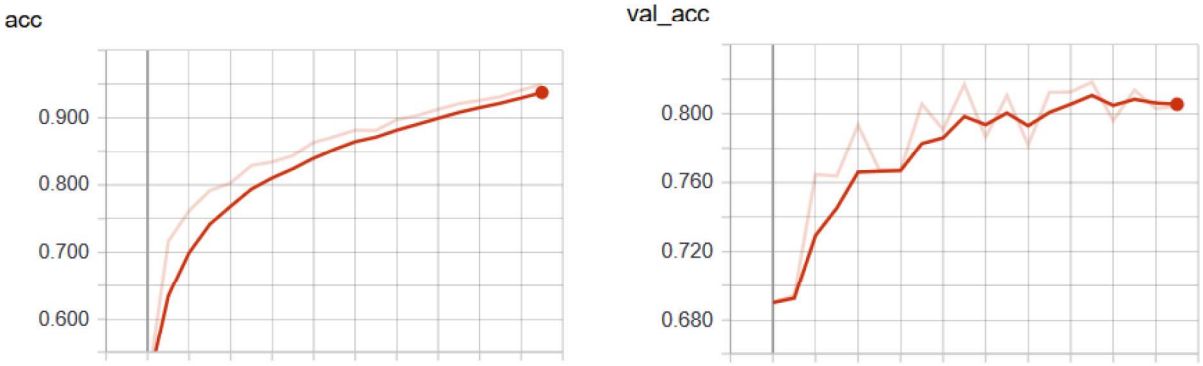
Batch size	512
Epochs	20
Validation split	0.2
Learning rate	1e-2
Momentum (Stochastic gradient descent)	0.9
Test accuracy	0.8538
Test loss	0.94691010305643082
Training time	09:36



7.3.2. VGG19

Graph/20190323-133619

Batch size	512
Epochs	20
Validation split	0.2
Learning rate	1e-3
Momentum (Stochastic gradient descent)	0.9
Test accuracy	0.7946
Test loss	0.8521510410428047
Training time	11:35

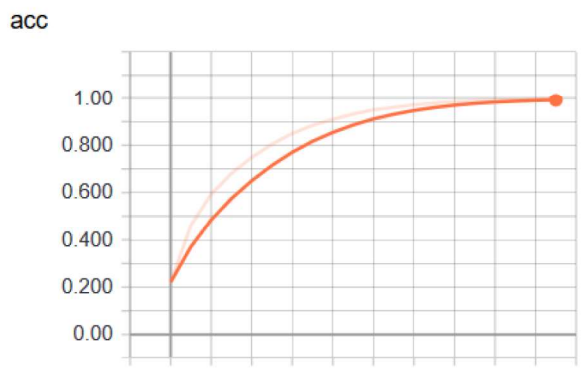


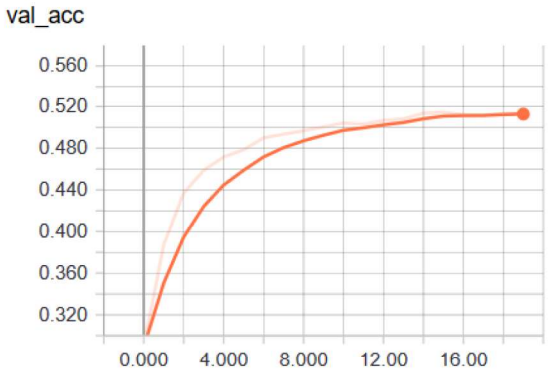
7.3.3. ResNet50

Graph/20190323-142757

Batch size	512
Epochs	20
Validation split	0.2
Learning rate	0.0005
Momentum (Stochastic gradient descent)	0.9
Test accuracy	0.5103
Test loss	2.011189196014404

Training time	12:40





7.3.4. Xception

Graph/20190323-144936 time 12:21

Batch size	512
Epochs	20
Validation split	0.2
Learning rate	1e-2
Momentum (Stochastic gradient descent)	0.9
Test accuracy	0.5861
Test loss	2.299872544670105
Training time	12:21