Deep Learning – Final Project

Prototypical networks and few-shot learning

# Authors:

* Michael Bikman - 317920999
* Nawras Abbas – 315085043

# Project goal(s)

Implement few short learning procedures using prototypical networks approach.  
  
The main goal of the project is to implement prototypical network with CNN (based on [1]) and then try to improve the results without using a significantly more complex architecture.

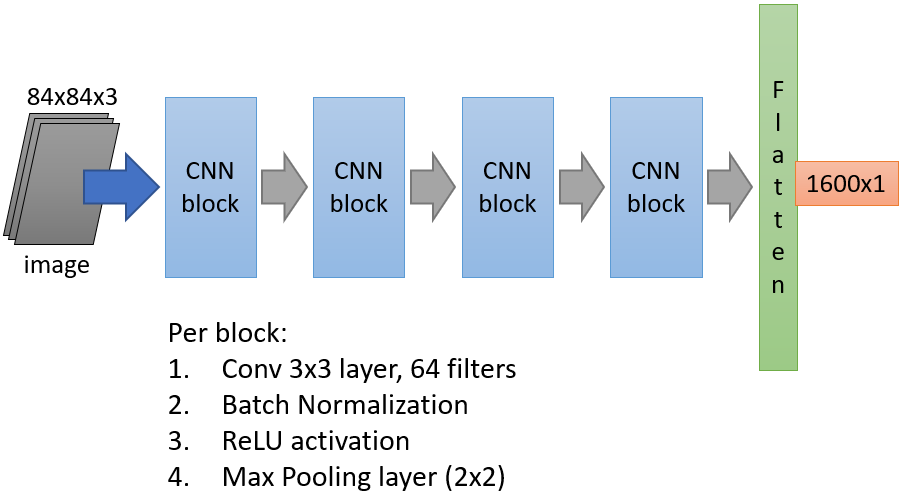
Steps we have planned:

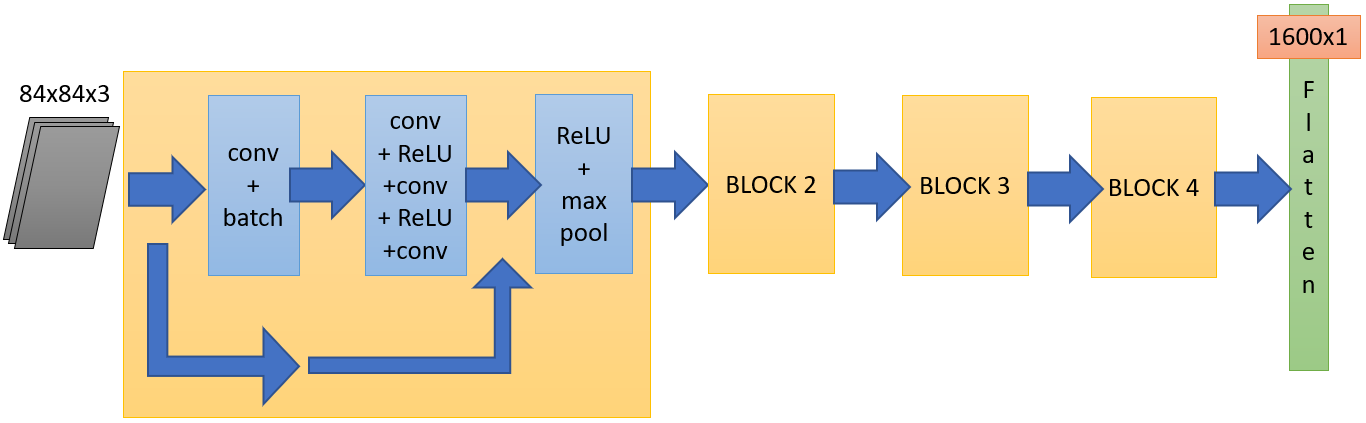
1. Create CNN for prototypical network following the description in [1]
2. Reproduce the results from the paper [1] – use it as a baseline
3. Tweak the CNN configuration to improve performance
4. Implement and use other possible networks
   1. Densely connected CNN
   2. Residual Network
   3. Residual Network with Stochastic Depth

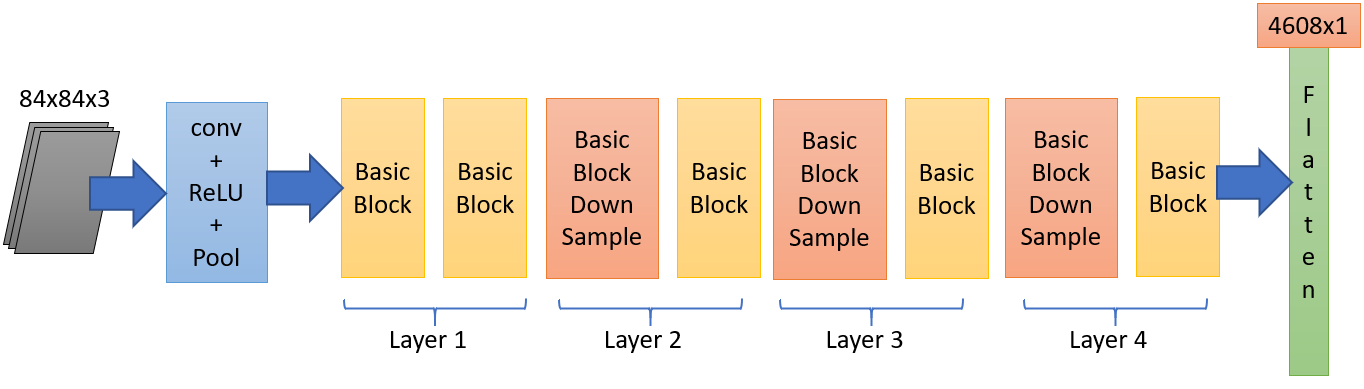
# Models architecture

In our project we used 4 different model of various sizes and types.   
(Please refer to [appendix 3](#_Appendix_3_–) to see models’ summaries).

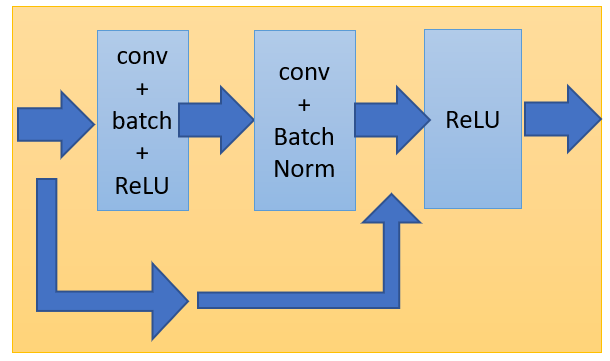
1. **Convolutional network with constant number of feature maps per block**

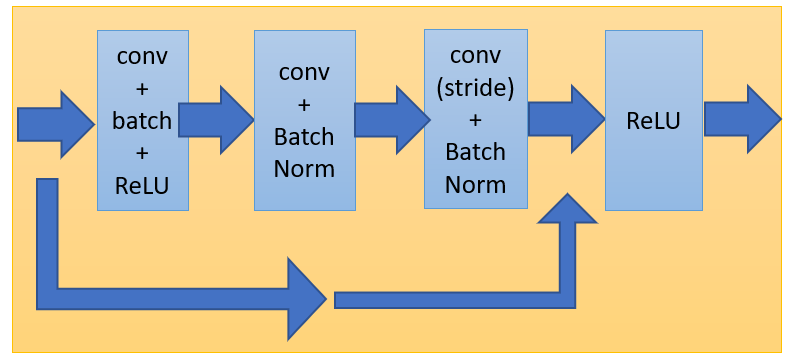
  
  
This architecture closely follows the setup from the original paper [1]

1. **Convolution network with dense layers**  
   This architecture is similar to previous one, but we have added residual connections and increased the number of convolutions. A very close architecture was used in [2].  
     
   Setup used:
   * Convolution 3x3 with 64 masks was used
   * Max pooling was 2x2
2. **Residual deep network based on ResNet18**

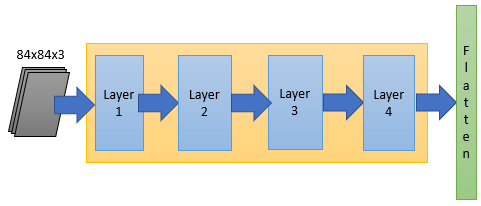
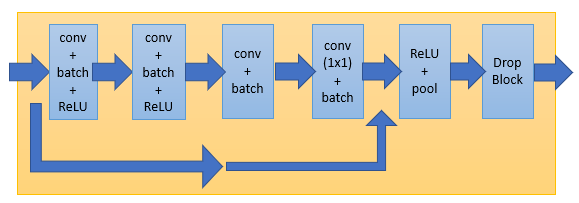
We have implemented the architecture on network identical to described in [3].  
  
Where:

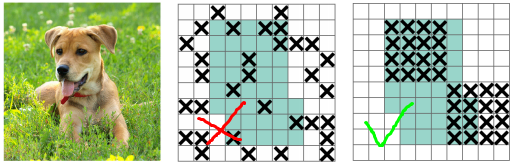
* Basic block is:



* Down sample block is the same as basic but with down sample stride convolution before activation function  
  

1. **Residual network with Drop Blocks**  
   We took the implementation of this residual network from   
   <https://github.com/corwinliu9669/Learning-a-Few-shot-Embedding-Model-with-Contrastive-Learning/blob/main/resnet.py>

(presented in article [9] and used in the paper [4]) and slightly simplified it - mainly by removing the “average pooling” part.   
  
Where each layer is defined by:  


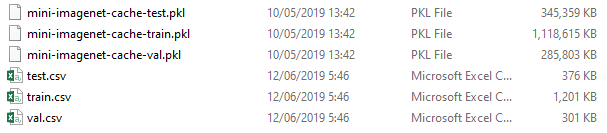
The main idea of this network – fight overfitting by dropping continuous regions from a feature map.   
Example:  
  
It is done by the drop block at the end of each layer (first layers are more preserved to save impact on the learning, while the last layers drop much more from feature maps).

# Data & Processing

## Dataset

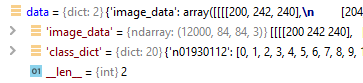
We used a very well-known mini-ImageNet dataset that contains pictures of size 84x84, joined into 100 classes (from [5])  
<https://www.kaggle.com/whitemoon/miniimagenet>

Splits

100 classes are divided into 64, 16, and 20 classes respectively for sampling tasks for meta-training, meta-validation, and meta-test, as proposed in [6]  
  
From Kaggle we have downloaded Python pickle files that already divided into test, validation and train datasets.  


No further preprocessing was made to images.

# Data Loading

1. Manual loading from files:  
   We used Pickle to load the data files. Each data file is loaded into dictionary with 2 keys  
   

* “*image\_data*”-> array of all pictures
* “*class\_dict*” -> dictionary of indices per class name

1. Episodes creation:  
   Each training episode contains of n-ways and k-shot, where “# ways” is the number of classes per episode, and “# shots” is the number of shots per class.  
   We have implemented a simple algorithm that randomly chooses defined number of ways and shots from data dictionary and thus creates an episode.

# Optimizations

1. Adam optimizer was our default choice, since it has shown the best speed and performance during learning. We also used the default settings for it.
2. Learning rate:

We used Step LR scheduler to reduce learning rate by 50% every defined number of steps.

# Experiments

### Training & Evaluation process

We have followed the standard training and evaluation procedure:

* 100 episodes per epoch
* 10K max epochs (in fact we have never got even to 1000 epochs)
* early stopping with patience 100 initially, later we have changed it to 200
* loss function was implemented according to [1] and all models use the same loss calculation
* learning rate started from 0.002 and reduced each 20 epochs   
  (later changed the steps value to 50)
* evaluation was made at 5-ways-5-shots episodes, while in training number of shots was different (under the restriction of GPU memory capacity)

### Model 1 – CNN

1. Our first milestone was reproducing the results from paper [1].   
   We have implemented the model 1 as CNN and tried a different number of ways in training:

|  |  |
| --- | --- |
| # ways (5-shot) | test accuracy (5-way-5-shot) |
| 10 | 65.96 0.99% |
| 14 | 66.54 0.8% |
| **16** | **66.90 0.69%** |

According to the researches, best performance was achieved when training with more ways, than in evaluation. In the paper they have got to 68.2, while using 20 ways per episode.   
Unfortunately, we have only 1 GPU (and not a very powerful one), so the maximum ways we have got is only 16.   
From the paper [1], for 5-way-5-shot:



In any case from now we consider 66.90% accuracy as our baseline and we try to improve it.

1. Adding the dropout.

Since overfitting is the worst problem we face in few-shot learning, we have tried to increase generalization by adding some dropout to the model.

|  |  |
| --- | --- |
| dropout (%) | accuracy |
| 0 | 66.540.8% (baseline) |
| 20 | 56.130.3% |

Clearly, it didn’t work.   
Not only we didn’t improve the score, but made it significantly worse. Looks like dropout was completely messing the score.

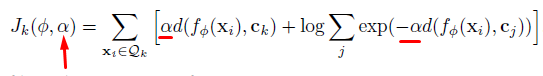
1. Mahalanobis distance vs. the Euclidean one.

Next idea was to try some other metric and we have chosen the Mahalanobis distance, which is a generalization over the Euclidian distance. We have implemented the calculations part and run the training, but it didn’t perform well.  
The calculation required to work with matrices of dimension 1600x1600 (according to the embedding size for our model), and even on CUDA we had severe raise in runtime.

|  |  |
| --- | --- |
| Distance metric | time per epoch (sec) |
| Euclidian | 17.7 (baseline) |
| Mahalanobis | 433.6 |

Since Mahalanobis distance was 25 times slower than the basic one, it was impossible to run for such a long time, and we have decided to try other methods.  
  
Note: we have tried to add some FC layers after the CNN to reduce output dimension and run calculations faster, but this version of network just stopped learning (output vector was not large enough to support proper learning).

1. Adding temperature parameter to distance calculation.

Authors of paper [2] suggested to use metric scaling to improve the accuracy of few-shot learning. Instead of just calculating the Euclidian distance we have added a learnable parameter (alpha) to be used as scaling factor in loss calculation   


* J = loss
* d = distance metric
* c = prototype
* f = embedding function
* Q = query set

We have tested the influence of metric scaling:

|  |  |  |
| --- | --- | --- |
| model | no scaling | with scaling |
| 64 maps, 10 ways | 65.96 0.99% | **66.40.5%** |

We indeed had the improvement of approx. 0.5% (not much, unfortunately), but we have decided to use the scaling for further steps.

1. Increasing number of feature maps.  
   Now we have tried to use more complex models (under the same architecture) by increasing the number of convolution maps in each stage. Due to hardware limitations we had to decrease number of ways for training to still be able to run without “out-of-GPU-memory” error.

|  |  |  |  |
| --- | --- | --- | --- |
| # of maps | # of ways | metric scaling | accuracy |
| 64 | 16 | yes | 66.900.7% (baseline) |
| 128 | 8 | no | 66.860.8% |
| 128 | 8 | yes | 67.690.3% |
| **128** | **16** | **yes** | **68.250.8%** |

It can clearly be seen that adding more maps allowed us to get better performance.  
Also we checked the influence of metric scaling on a model with more maps and saw even a better improvement.  
Note: model with 128 feature maps and 16 ways was trained in Google Collab to avoid failure with GPU errors.

1. Fine tuning the learning rate

And last, but not the least we have tried several learning rates and scheduler step values to tune the accuracy to the maximum

* on a model with 128 maps and 8 way per training episode
* each time the LR decreased by 50%

|  |  |  |
| --- | --- | --- |
| learning rate | step value | accuracy |
| 0.01 | 20 | 67.241% |
| 0.001 | 20 | 65.870.4% |
| 0.002 | 20 | 67.690.3% |
| **0.002** | **50** | **68.361%** |
| 0.002 | 100 | 67.490.6% |

After all, we have increased the accuracy percentage and found the maximum performance improvement on this model vs. baseline (68.36% vs. 66.9%) just by adding more masks and tuning the learning rate (plus some tiny minor metric scaling).

Note: improved model has more parameters and thus training time has increased as well

|  |  |  |
| --- | --- | --- |
| # feature maps | # trainable parameters | time per epoch (sec) |
| 64 | 113,088 | 17.7 |
| 128 | 447,360 | 31.3 |

### Model 2 – Dense layers CNN

After getting the improvement with regular CNN model we have decided to add some residual connections to create a “dense” model.

Unfortunately, the performance was awful – only around **50%**.  
  
This model was an unsuccessful try, but still we wanted to continue with other residual networks.

### Model 3 – Residual network

For this model we took the classical ResNet18 architecture (from paper [3]).

|  |  |  |
| --- | --- | --- |
| # ways | dropout | accuracy |
| **16** | **no** | **60.00.5%** |
| 16 | 0.2 | 52.50.5% |

Once again, we have tried to add a dropout (after ReLU), but still we saw the same familiar effect of reducing the performance. On the other hand, the accuracy without the dropout was around 60%, that was promising.  
Here the number of parameters was huge (11,176,512) – and we thought that it might be a good idea to look for the smaller (much smaller) residual network.

### Model 4 – Residual network with drop blocks

We have found architecture (ResNet12) used in [8], following the practice of papers [2] and [7].   
To save time, we simply took the ready implementation from the GitHub <https://github.com/kjunelee/MetaOptNet>   
removed the last averaging layer and tried various tweaks.  
(We could run the network only with 5 ways per train episode)

1. Changing the number of feature maps.

In original paper, the researchers have used [64, 160, 320, 640] maps configuration. We have tried several other options, and only with 5-ways per train episode, since all other values failed to run under GPU restrictions.

|  |  |
| --- | --- |
| # maps per layer | accuracy |
| [64, 160, 320, 640] | 61.150.6% |
| [64, 128, 128, 256] | 63.160.9% |
| [32, 64, 128, 256] | 62.600.6% |
| **[64, 64, 64, 64]** | **64.880.8%** |

Surprisingly the configuration with 64 maps performed the best (similar to CNN model 1 with the same number of maps, with 66.9%).   
Number of parameters for [64, 64, 64, 64] maps is 421,760.

1. Different drop block rate.

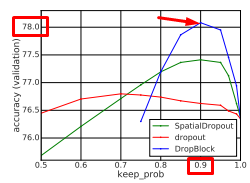
This architecture allows to remove some areas from feature maps (with certain probability) during training. It was introduced in [9] and shown that it outperforms other techniques like regular dropout, cutout and others.  
Default drop block rate was set to 0.1, but we have tried also 0.5 (using the [64, 64, 64, 64] maps setup).

|  |  |
| --- | --- |
| drop block rate | accuracy |
| 0.1 | 64.880.8% |
| 0.5 | 58.530.2% |

Higher drop rate reduced the accuracy (as we predicted, but the score was lower than we expected).  
We have checked the root cause of low performance and that led us straight to the main improvement.

1. Removing nodes dropout completely.  
   One of the features of this architecture was applying regular dropout on neurons if block dropout was not applied.   
   But for all previous experiments we have clearly seen that nodes dropout dramatically reduces the accuracy, so we decided to remove it – and it raised the score significantly!

|  |  |
| --- | --- |
| drop block rate | accuracy |
| 0.0 | 70.511.1% |
| **0.1** | **70.891.2%** |
| 0.2 | 70.391.0% |

Interesting to mention, that in [9] authors also got the best performance on ImageNet with drop rate 0.1 (or keep probability 0.9 which is the same as drop 0.1).  


# Results

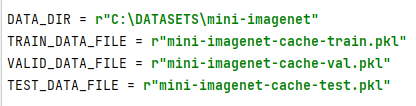
According to our experiments, best result is achieved with:

* The 4th Model (Residual network with drop blocks)
  + Using temperature distance parameter
  + And with some dropping of network residual blocks during training

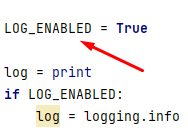
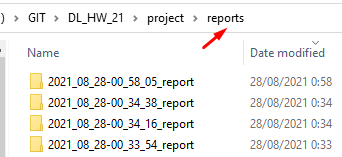
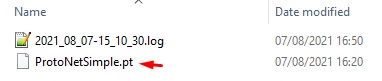
It is worth to mention that the best Model 4 (residual), uses less parameters (421K vs. 447K) than the best Model 1 (CNN with 128 maps), and achieves better performance (70.9% vs. 68.36%), while consuming less GPU resources as well.

Conclusion  
We have reproduced the baseline results (nearly) and then succeeded to improve the accuracy on “5-way, 5-shot” episodes only by increasing the number of convolutional feature maps, adding learnable parametric distance instead of simple Euclidian metric, and finally switching to the residual network with drop blocks, which gave us almost 4% improvement over the baseline (66.9% vs. 70.89%).  
  
Of course, during the last few years there were many excellent and outstanding results achieved on this particular problem, with state-of-the-art performance now is more than 80%, but all these solutions have used a much more complex architecture and setup.  
  
Our small project was to figure out how much we can improve the performance only by choosing an appropriate network model and finely tuned parameters.

# Appendix 1 – Files attached and data paths

TBD  


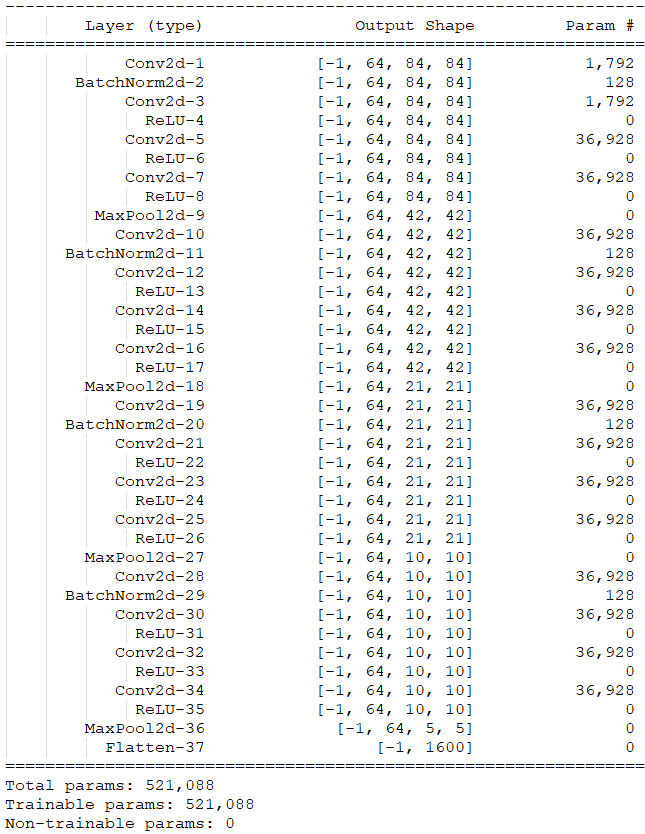
# Appendix 2 – Logging

Logging to file can be enabled / disabled by the flag.  
  
All the logs are created in ‘reports’ directory  
  
The log location can be configured.  
  
Each report folder for train also contains the best model saved during training.  


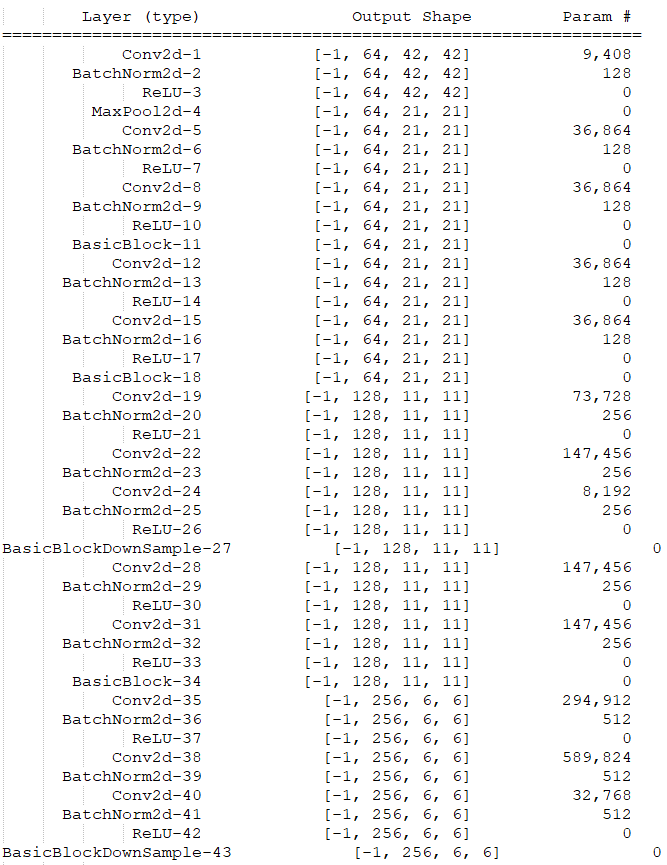
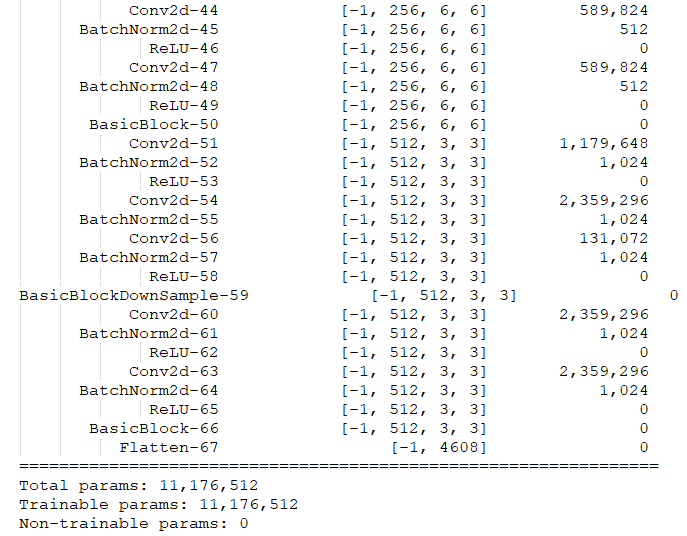
# Appendix 3 – Models summaries

## Model 1 – CNN

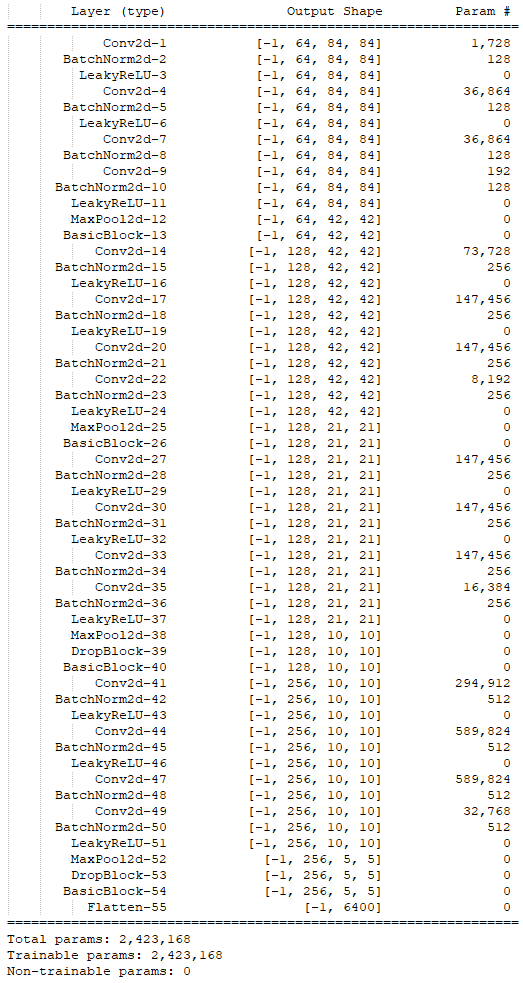
## Model 2 – Dense layers CNN



## Model 3 – Residual network based on ResNet18

Model 4 – Residual network based on resnet12

(configuration with 64, 128, 128, 256 maps)  


# Appendix 4 – Papers and material used

1. “Prototypical Networks for Few-shot Learning” - <https://arxiv.org/abs/1703.05175>
2. “TADAM: Task dependent adaptive metric for improved few-shot learning” - <https://arxiv.org/abs/1805.10123v4>
3. “Deep Residual Learning for Image Recognition” - <https://arxiv.org/abs/1512.03385>
4. “Learning a Few-shot Embedding Model with Contrastive Learning” - <https://ojs.aaai.org/index.php/AAAI/article/view/17047>
5. “Matching Networks for One Shot Learning” - <https://arxiv.org/abs/1606.04080>
6. “Optimization as a model for few-shot learning” - <https://openreview.net/pdf?id=rJY0-Kcll>
7. “A Simple Neural Attentive Meta-Learner” - <https://arxiv.org/abs/1707.03141v3>
8. “Meta-Learning with Differentiable Convex Optimization” - <https://arxiv.org/abs/1904.03758v2>
9. “DropBlock: A regularization method for convolutional networks” - <https://arxiv.org/abs/1810.12890>

# Appendix 5 – Links

1. <https://paperswithcode.com/dataset/miniimagenet-1>
2. <https://www.machinelearningplus.com/statistics/mahalanobis-distance/>
3. <https://www.borealisai.com/en/blog/tutorial-2-few-shot-learning-and-meta-learning-i/>
4. <https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035>
5. <https://towardsdatascience.com/residual-network-implementing-resnet-a7da63c7b278>
6. <https://pythonrepo.com/repo/KevinMusgrave-pytorch-metric-learning-python-pytorch-utilities>