POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

CIE6006 - Data Analytics

Assignment 1

COMPUTER SCIENCE AND ENGINEERING - INGEGNERIA INFORMATICA

Author: Tito Nicola Drugman

Professor: Prof. Qingyu Li

TA: Weiwen Yuan, Junsheng Yao, Guanlin Wu

Academic year: 2025-2026

---

1. Introduction

This document is the report for the first assignment of the *Course CIE6006/MCE5918 - Data Analytics*.

2. Batch Normalization Implementation

2.1. Introduction

Batch Normalization [2] addresses the issue of "internal covariate shift", where the distribution of each layer's inputs changes during training, complicating the learning process. The core idea is to normalize the pre-activation outputs of a layer for each training mini-batch.
Specifically, for a given layer and a mini-batch of N samples, the output of the preceding affine transformation is a matrix of shape (N, D), where D is the number of neurons in the layer. Batch Normalization operates on this matrix column-wise. For each neuron's output vector (a column of dimension N), it calculates the mean and variance across the mini-batch. These statistics are then used to normalize the neuron's outputs to have a zero mean and variance equal to one.
The normalization is followed by a learned affine transformation, where a scale parameter (γ) and a shift parameter (β) are applied. This allows the network to learn the optimal mean and variance for each layer's inputs, rather than being strictly constrained to a standard normal distribution.

2.2. Effects of weight initialization

Figure 1 reports three plots. **Plot 1** shows the best value accuracy on the y-axis and the weight initialization scale on the x-axis and it compares the baseline against the batch normalization. The baseline (plotted in blue) is extremely sensitive to the weight scale. For small scales (up to 10^-3) the best validation accuracy is around 12.5%. It then shoots up to a peak close to 10^-1 reaching around 30% to then suddenly drop to 15% for larger scales. The batchnorm (plotted in orange) is more robust than the baseline. It is able to achieve a validation accuracy around 25% across a wide range of weight initialization. The performance of the batchnorm is almost always superior to the baseline. Similarly to the baseline the performance starts to degrade significantly at very large scale.
Plot 2 compares the weight initialization scale to the to the best train accuracy. This plot mirrors the validation accuracy plot and we can see the two plots together in Figure 2. The baseline

model is able to learn effectively only in a narrow band of initialization close to 10^1, while the batchnorm model learns well across almost the entire range and achieves higher training accuracy than the baseline for almost all the weight initialization selected. **Plot 3** show the final training loss compared to the weight initialization scale. For the baseline the loss stays around 2.3 for small weight scale. For batchnorm the loss is consistently low across a very broad range of scale.

Results and Observations The baseline model is highly sensitive to the initial weight scale and is able to achieve meaningful accuracy only within a very narrow range of scale, while for initializations outside the optimal window the model fails to train effectively, thus resulting in poor accuracy. On the other hand, the model with batch normalization shows more robust results to the initialization scale and consistently achieves strong validation and training accuracy across a wide spectrum of scale.

A network without BN is more prone to vanishing or exploding gradients, in particular if the initial weights are too small the signal propagating through the network tend to zero without contribute to learning, while if the weights are too large the signal can grow exponentially leading to numerical instability. By normalizing[1] the output of each fully-connected layer BN guarantees that the input to the next activation function have a stable and well behaved distribution and thus prevents the signal from vanishing or exploding and also becoming less sensitive to the initial scale of its parameters.

Interesting consideration It it interesting to notice that in Figure 1 and Figure 2 for all the plots the baseline perform better than the batchnorm when the weight initialization scale is exactly 10^1. This happens because it is a sweet spot that is ruined by BN since the mean and variance are calculated on a mini-batch and not on the full dataset and this introduce a small amount of noise into the training process and thus the BN model can not reach the same peak performance as the perfectly-tuned baseline. Of course we need to keep in mind that this experiment is limited to ten epochs, probably the BN model might surpass the baseline due to its better stability over the long

[1] mean 0 and variance of 1 for every mini-batch

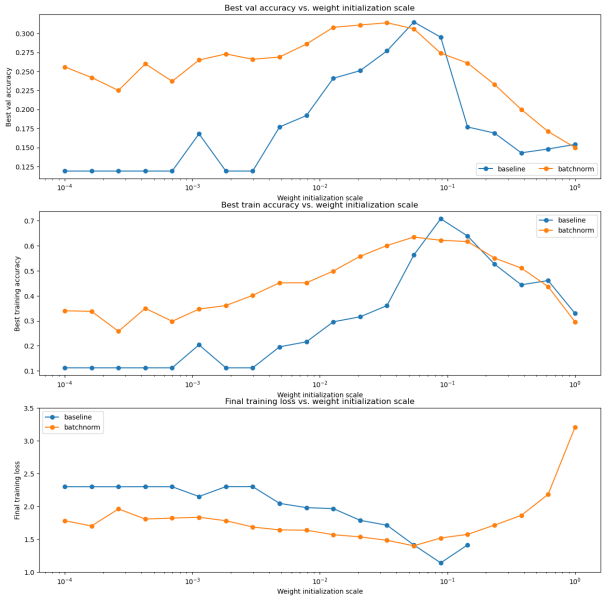

Figure 1: Interaction of batch normalization and weight initialization.

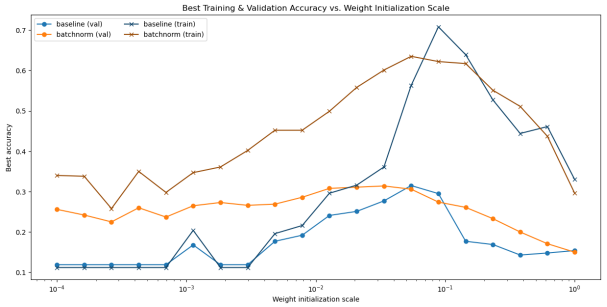

Figure 2: Interaction of batch normalization and weight initialization.

run.

2.3. Effects of Batch Size

In Figure 3 we can see a comparison between the baseline (with pink triangles) and different batch sizes (5, 10, 25, 50, 100 and 200).

In **Plot 1 - Training Accuracy** we can see a general trend where larger batch size leads to better training. The red line (BN50) consistently achieves the highest training accuracy surpassing 80% by epoch 10. Smaller batch size achieve significantly lower training accuracies, in particular BN5 (blue) underperformed with respect to the baseline. Smaller batch size have a higher variations while bigger batch sizes show a more smooth and stable learning process that keep improving as the number of epochs

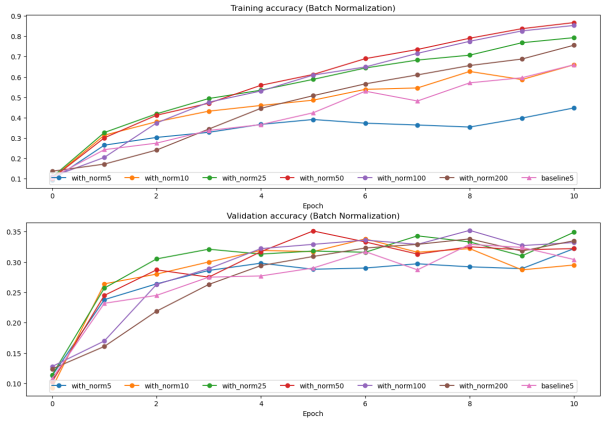

Figure 3: Interaction of batch normalization and batch size.

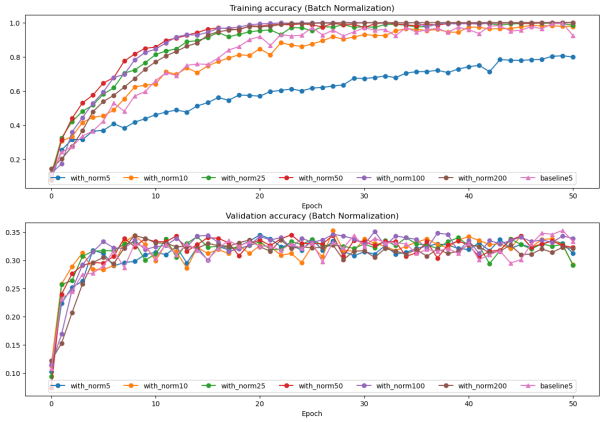

Figure 4: Comparison of several possible batch sizes over a more extended number of epochs.

increases.

In **Plot 2 - Validation Accuracy** the trend is similar but less pronounced. The baseline and smaller batch size perform worst while models with larger batch sizes achieve higher validation accuracy peaking around 30%. While we could expect that larger batch sizes lead to better result this is not always the case since batch sizes of 25, 50, 100 and 200 are all around 32-35% mark.

Extending the training to 50 epochs (Figure 4) reveals clear evidence of overfitting across all models. BM5 also seems to heading to overfit as well, but more slowly. What is interesting to notice is that all the models reach a validation accuracy around 33% close to epoch 15 without any improvement. We can see in the validation plot that there is no clear winner among the different batch sizes after the initial training phase, all the different batch sizes seems to perform similarly with their accuracies criss-crossing each other while remaining around 30%-35%. Even the baseline and BM5 perform within this same band. This suggests that for this specific problem and architecture, once the model starts to overfit, changing the batch size for Batch Norm has little effect on its ability to generalize.

2.4. Comparison of Batch Normalization and Layer Normalization

To answer *Inline Question 3* of the Batch Normalization Jupiter Notebook *Option 3* states we should subtract the "mean image" of the dataset from each image. To calculate this "mean image", one must average the pixel values for each specific pixel location across the entire dataset. The dataset here acts as one very large batch. This process of using a shared, dataset-wide statistic to center every individual image is directly analogous to Batch Normalization's use of shared, batch-wide statistics. However, we must also divide by the standard deviation to guaranteed a standard deviation of one.

Option 2 describes scaling each image so that the sum of all its pixel values equals 1. This calculation is performed entirely within a single image. You take one image, compute a statistic based on all its pixels and use that statistic to normalize that same image. The process doesn't require information from any other image in the dataset.

2.5. Underperformance of Layer Normalization

To answer *Inline Question 4* of the Batch Normalization Jupiter Notebook we need to remember that layer normalization works by computing the mean and standard deviation across all the features for a single data point. If the input has many features then layer normalization will be able to compute a stable and meaningful mean and standard deviation, but if instead we have a small number of features (or short width and height of the image) then the statistics we obtain will be noisy and unstable and the normalization might harm the learning process (*Option 2*).

3. Dropout

Dropout[1] is a powerful regularization technique designed to combat overfitting in neural networks. The core idea is to prevent complex co-adaptations where a neuron becomes useful only in the context of specific other neurons being active. This over-specialization can lead to poor generalization on test data. The idea behind dropout is that during each forward pass in the training phase, a random portion of neuron outputs in a layer are "dropped" (set to zero) with a certain probability. This means that for every training example, the network is effectively thinned into a different sub-network. Because a neuron can no longer rely on its neighbours being present, it is forced to learn more robust features that are useful on their own, across a wide variety of internal network contexts.

3.1. Backward Pass and Scaling

During training the inverted dropout perform two tasks:

- **Masking**: which means we multiply x [2] by a matrix of the same shape which contains only 0s and 1. The probability that any single element in the mask will be a zero is equal to *1-p*.
- **Scaling**: the result is then multiplied by the keep probability p which scale up the activations that were not set to zero.

To summarize if we select a single neuron that survives (mask is 1) then the operation is $out\_survivor = x\_survivor \,/\, p$, this means that the valuation of x_survivor has been amplified by a factor of 1/p.
If we fail to divide by p during the backward pass the calculated gradient would be smaller than the true gradient by a factor of p and this smaller error signal would be passed to all preceding layers producing a smaller weighted update and slowing down the learning process. As a result the network would fail to train effectively because the gradients would not be scaled correctly to account for the amplification occurred during the forward pass.
In Figure 5 it is possible to see the importance of scaling. The graph show two identical networks with keep probability set to 0.5. The blue network uses the correct inverted dropout implementation, while the second uses an incorrect version without the scaling *1/p*. Even if both models show a similar learning for the first few epochs there is a clear difference between the two networks since the model with correct scaling consistently achieves a higher training accuracy reaching a plateau around 39%. On the other hand, the model with no scaling strategy achieves a lower accuracy of approximately 35%. In the validation accuracy plot we can see the same trend of the training accuracy and the correctly scale model maintain a superior validation accuracy as the epochs increases.

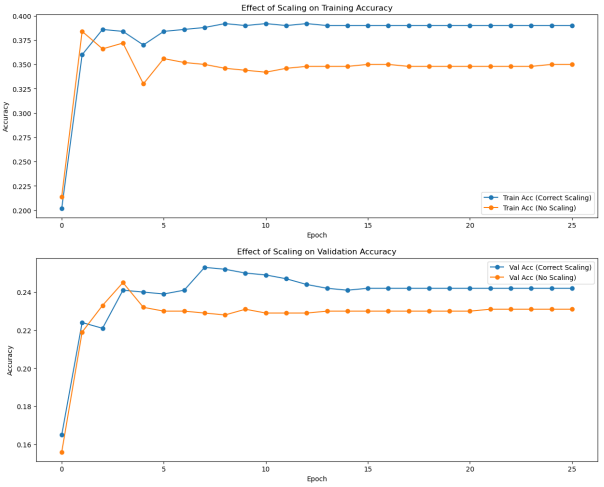

Figure 5: The importance of scaling in dropout.

3.2. Dropout as regularizer

In Figure 6 the training accuracy of the model without dropout (blue dots) increase quickly and get close to 100%, while the training accuracy of the model with dropout (orange dots) also increases but is consistently lower than the no-dropout model's accuracy. The dropout model is learning more slowly and never reaches 100% of training accuracy, this is because a portion of its neurons are randomly shut off this making the learning task more complicated. In the second plot of Figure 6 we can see that the no-dropout model's performance is lower even though it had near-perfect training accuracy (overfitting). On the other hand the dropout model achieves a consistently higher validation accuracy, showing that what the model learned during its more difficult training was more robust and generalized better to unseen data.

[2] x is the matrix of activations values coming from the previous layer

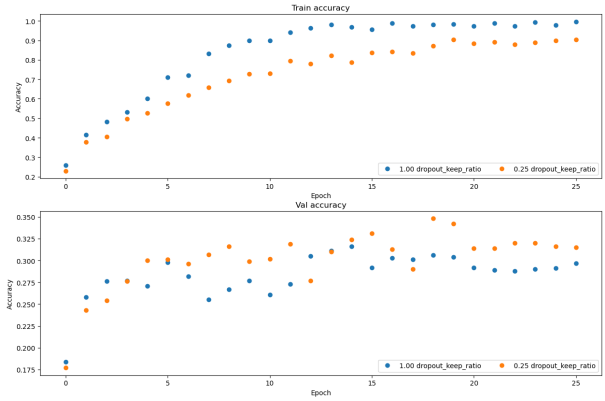

Figure 6: Dropout as regularizer.

3.3. Different values of dropout

In Figure 7 I decided to run an experiment with varying levels of dropout strength by training several identical networks with p values ranging from 1 (no dropout) to 0.25 (strong dropout).In the validation accuracy plot we can see that the model with no dropout perform relatively poorly, similarly the models with weak dropout (0.75 and 0.66) show close accuracy. The best models are the one with a stronger dropout (0.5, 0.33 and 0.25) even if they suffer from a high variability. This tiny experiment confirms that dropout is a critical hyperparameter that control the balance between overfitting and underfitting and the best approach, if possible, is to find the best value with hyperparameter tuning or trial and error. The sweet spot seems to be around 0.5 but further experiments and longer training are required to prove this.

4. Convolutional Network Implementation

When a convolutional neural network model is fully trained, even if it is consider a black box we can still obtain some valuable[3] insights by visualizing the weights of the first convolution layer. Obviously the raw weights value can be positive or negative, to make them visible (and meaningful[4]) to use we need to normalize them by shifting and scaling the values for each filter so that the smallest value is mapped to 0 and the largest is mapped to 255. Usually the first layer learn to detect simple and fundamental patterns in the raw image data since their input is the natural image itself, technically speaking we could also visualize other layers but those layers do not have as input the original image and they tends to be high-dimensional tensors. If we try to visualize the weights of a second-layer filter it would look like random noise to us.

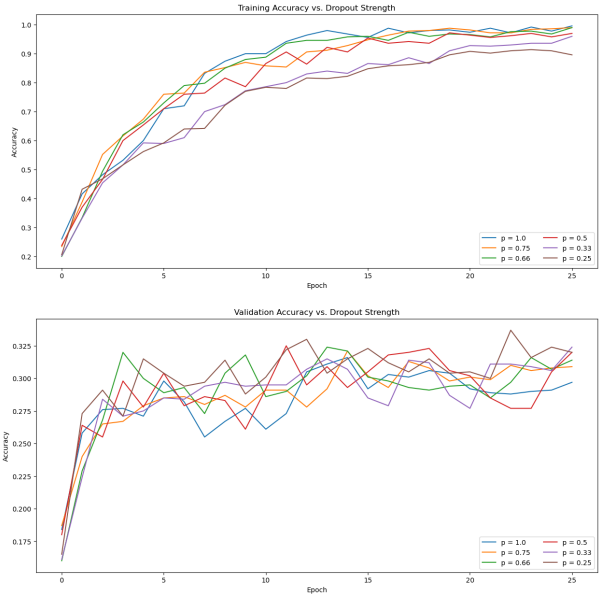

Figure 7: Comparison of training and validation accuracy for different dropout keep probabilities.

In Figure 8 it is possible to see the first layer's weights, the size of each of the 32 filters is 7x7x3 and the total dimension of the first layer's weight tensor is (32,7,7,3). It is possible to see that many filters have learned to act as edge detectors, as an example several filters show a sharp transition from a dark area to a light one or from one colour to another, which makes those filters sensitive to edges and various angle and orientations. Furthermore some filters show specific colour or colour contrasts which show that those filters have possibly learned colour detection. We can see that even if we consider CNN as a black box, the model is learning a representation of the world that (in some cases) we can analyse and we can find some meaning related to what the model learned. This CNN has inde-

[3] it can be argued that those insights are not really valuable for task-specific understanding. This is because, regardless of the task, the first layer's weights converge to a set of generic, low-level feature detectors like edges and simple color-blob detectors. These filters are simply learning the fundamental statistical regularities of natural images, which is a prerequisite for any vision task, rather than revealing anything unique about the classification problem itself.

[4] read previous footnote.

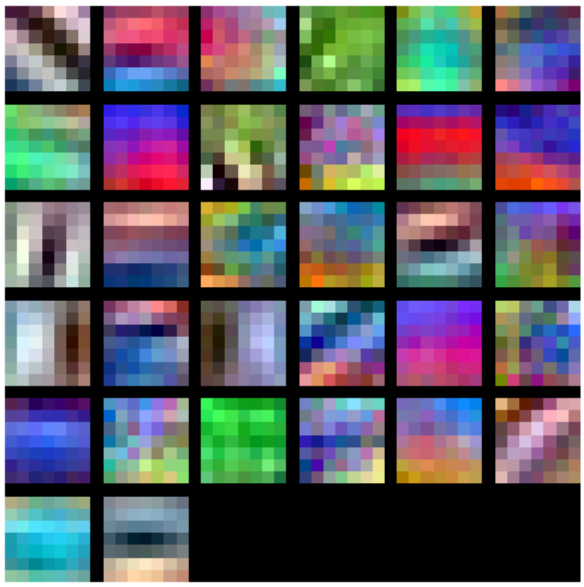

Figure 8: Visualization of the first layer's filters.

pendently discovered the importance of detecting fundamental visual primitives such as edges and colours which are the building blocks that subsequent layers will use to identify more complex shapes and ultimately (maybe) the object classes themselves.

5. Pytorch Implementation

This section was run locally on a workstation located in Italy and access remotely thanks to Windows Desktop Remote. The workstation has a Dual NVIDIA RTX 3060 Ti GPUs (16GB), 16-Core CPU and 128GB RAM.
To achieve high accuracy on CIFAR-10[3] I decided to implement and compare two deep convolutional neural network architectures. Both models uses the same training augmentation techniques which involves horizontal flipping and random crop with padding. Furthermore I used AdamW optimizer which is similar to Adam but also handles weight decay more effectively. A CosineAnnealingLR scheduler was used with an initial learning rate of 1e-3 and a weight decay of 1e-4. This dynamically adjusts the learning rate over 15 epochs, allowing for large updates early in training and fine-tuning towards the end. Lastly I trained both models across multiple GPUs (nn.DataParallel) to speed up the training process.

5.1. Model 1 - VGG style model

This model uses a deep feature extractor with three blocks (Conv-BN-ReLU-MaxPool) with the number of filters increasing from 64 to 128 and then to 256. After the convolutional layers the feature map is flattened to be fed into a linear layer of 1024 units followed by a final output layer.

5.2. Model 2 - GAP (Global Average Pooling) model

This model use the same three block convolutional base of the VGG style model but instead of a fully connected it uses AdapativeAvgPool2d to reduce the feature map to a single value. This results in a smaller vector of length 256 that is directly fed into the final output layer.

5.3. Results

The VGG style model significantly outperfromed GAP model achieving a peak validation accuracy of 88.20% compared to the GAP model's 80.40%. This suggest that for a CIFAR-10 complexity the additional parameters of VGG are highly beneficial. In Figure 9 it is possible to see that in the training plot the VGG-style model's loss decreased much faster and reached a lower value with respect to the GAP model. This indicates that VGG was able to learn the underlying patters in the data more efficiently. Furthermore VGG-style model achieved a 87.19% accuracy on the test set which is very close to the validation accuracy.Both models surpassed the (low) threshold of 70% this result can show that combining a deep architecture with techniques like AdamW, learning rate scheduling and batch normalization is a powerful and reliable strategy for training deep networks. The VGG-style model was the clear winner. Figure 9 shows it not only reached a much higher peak accuracy but also learned faster, reaching 75% accuracy by epoch 2, while the GAP model took until epoch 7 to reach a similar level. Furthermore the VGG model's loss consistently remained lower than the GAP model's thus proving that it was more effective at fitting the training data. While the GAP architecture is more parameter-efficient and a strong regularizer, its limited classifier capacity was a bottleneck on this task.
I decided to analyse the trade-off of both ar-

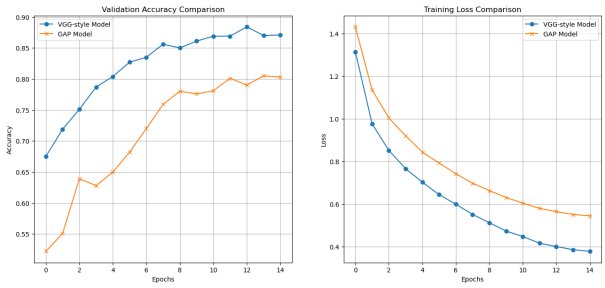

Figure 9: Comparison of VGG-style model and GAP model

chitectures by also evaluating their computational complexity and memory requirements. The results can be seen in Table 1. Since the benchmarking tool that I commonly use called ST Edge AI Developer Cloud[5] requires a different format for the model, the original PyTorch architectures were re-implemented in Keras. MACC (Multiply-Accumulate) operations depends only on the architecture and not on the framework, also ROM size should be similar since the parameter count will be the same and RAM size (is determined by the model parameters plus the size of the activation maps during a forward pass). It is possible to see that VGG-style model's superior accuracy comes at a significant cost. VGG has over more than 12 times the number of parameters of the GAP model, also its ROM footprint is consequently 12 times larger thus making it unsuitable for many memory-constrained microcontrollers. Furthermore VGG model has slightly more MACCs, this is because the majority of computations in both models occur in the shared convolutional base. Lastly it is interesting to see that both models have a nearly identical RAM footprint. This is because RAM usage is dominated by the largest activation map, which occurs in the early convolutional layers shared by both architectures.

This analysis show a classic trade-off: for applications where maximum accuracy is the most important goal and memory is not a constraint the VGG-style model is superior, but for edge AI and embedded applications where ROM size is a critical bottleneck the GAP model offers highly compelling alternative by providing reasonable accuracy with a smaller memory footprint.

[5] https://stedgeai-dc.st.com/home

Table 1: Comparison of Computational and Memory Metrics

Metric	**VGG-style Model**	**GAP Model**
Peak Validation Accuracy	**88.20%**	81.80%
MACC (Operations)	43.95 M	39.75 M
Parameters	4.58 M	0.37 M
ROM (Flash) Footprint	18.32 MB	1.51 MB
RAM (Activations) Footprint	97.28 KB	97.02 KB

6. Additional Work

Some additional work I have done is provided in past sections. Such as the graphs reported in Figure 4 which extends the training to 50 epochs and compute the training and validation accuracy for several identical models with batch size ranging from 5 to 200.

Additional graphs such as the ones in Figure 7 compare different dropout values ranging from 1 to 0.25 to evaluate how they affect the training and validation accuracy.

As part of the extra credit I also compared two network architectures VGG-style CNN and Global Average Pooling (GAP). More details are visible in Section 5, Figure 9 and Table 1.

References

[1] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[2] Sergey Ioffe and Christian Szegedy. Batch

normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[3] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009. Technical Report.