Tiny ImageNet Challenge Investigating the Scaling of Inception Layers for Reduced Scale Classification Problems

Emeric Stéphane Boigné
eboigne@stanford.edu

Jan Felix Heyse
heyse@stanford.edu

Abstract

Scaling of inception modules for reduced size classification problem is investigated and a smaller architecture for 200 *label datasets is proposed. It is inspired from the original GoogleNet designed for* 1000 *label datasets. The proposed architecture involves four successive inception modules and is trained using a dataset obtained through some data augmentation techniques performed on the Tiny ImageNet. The impact of learning rate and weight decay, two key hyperparameters on the learning process, is analyzed, and suitable range of values are identified. However, the proposed architecture is prone to overfitting. Tuning the number of parameters and using dropout yields better generalization without loss of performance for a validation accuracy around* 45%*. Test accuracy reaches* 41.3%*. Further emphasis of this study is made on analyzing more thoroughly our architecture and confirming its proper training. A first analysis is made in the feature space, right before the last fully connected layer, using dimensionality reduction techniques. A Principal Component Analysis and a better performing t-distributed Stochastic Neighbor Embedding are used to show clustering of classes in reduced 3D space. A second analysis focuses on visualizing features from intermediate layers through a mapping to the original image pixels, using backpropagation and Deconvolutional Neural Networks. Both techniques yield insightful images, which are used to discuss the performance of our architecture.*

1. Introduction and Problem Statement

The Tiny ImageNet is a database used for the training and testing of neural networks for visual recognition problems. It comprises a training set of $100,000$ labeled images, a validation set of $10,000$ labeled images, and a test set of $10,000$ unlabeled images with 200 different classes evenly distributed among each of the provided sets. The images are of size 64×64 pixels in gray scale and in RGB color space.

This dataset is used for a competition within the CS 231N class. The goal of this competition and of this project is to design and train the Convolutional Neural Network (CNN) that achieves the highest test accuracy.

The present study emphasises on the understanding of what our CNN is learning. To do so, we use different visualization techniques that provide some insight into the feature representation at different stages in the network.

2. Technical Approach

2.1. Dataset

The training data is first normalized so that it has zero-mean and unit variance. The validation and the test sets are pre-processed with the same normalization values as the training set. Approximately 2% of the data is not colored, and for simplicity we initially omitted these gray scale images. Later, we converted them to 3 channel images with each channel containing the gray scale values.

We used data augmentation techniques to increase the amount of available training data. Specifically, we added for each RGB training image a second one, which was with equal probability either

- flipped in the horizontal direction,
- rotated clockwise by 6, 8, or 10 degrees, or
- rotated counterclockwise by 6, 8, or 10 degrees.

The resulting augmented training set had $198,179$ images, almost evenly distributed over the 200 classes.

2.2. Architecture

For the architecture of our CNN, we looked at innovative and competitive neural networks from recent ImageNet Challenges [1]. In particular, the usage of parallel layers in so-called *inception modules* as proposed by Szegedy *et al*. [6] is a concept that we are experimenting with. We also used *batch normalization* before all activation functions, an idea first described by Szegedy and Ioffe [4].

2.2.1 Inception Modules

Inception modules are elements in CNNs that consist of parallel layers which are concatenated depth-wise. We used

parallel convolutions with spatial filters of size 1×1, 3×3, and 5×5 as well as an average pooling layer. The different filter sizes and their corresponding receptive fields allow for identification of features over a wider range of sizes.

Furthermore, our inception modules involved some dimensionality reduction in order to reduce the complexity of the model and the computational costs: The 3×3 and 5×5 convolutional layers are preceeded by 1×1 convolutional layers that reduce the depth of the ingoing data. Similarly, the average pooling layers are followed by such 1×1 convolutional layers.

The structure of the inception module including dimensionality reduction is illustrated in figure 1.

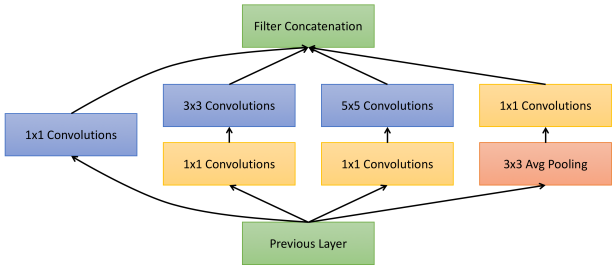

Figure 1: Inception module architecture

It is important to point out that each convolutional layer in- and outside the inception modules is followed by a batch normalization and a ReLU activation layer, in order to provide for non linear operations within the model.

One further advantage of the inception modules is that they are relatively efficient in terms of memory and operations needed, in particular when compared to the similarly well performing VGG neural network by Simonyan and Zisserman [5], which does not involve any parallel elements but sequentially stacked a larger number of convolutional layers of filter size 3×3.

2.2.2 Batch Normalization

Batch normalization is a technique developed to alleviate the problems resulting from bad weight initialization. It works by mapping the input into a desired range by specifying a learnable mean and variance for the output. Batch normalization layers are commonly used before activation layers where they can be interpreted as a pre-processing of the input data.

2.2.3 Overall Architecture

The overall architecture of our CNN is presented in this subsection. As GoogleNet was the first CNN to take advantage out of inception modules we oriented ourselves at their design when developing our own architecture. While GoogleNet was tuned to work with 1000 different classes, our task was categorizing into 200 classes, and hence we decided to reduce the number of parameters accordingly for our model.

Two different architectures are presented: a baseline one, and a slightly improved one. They share a common high level structure that is presented in figure 2.

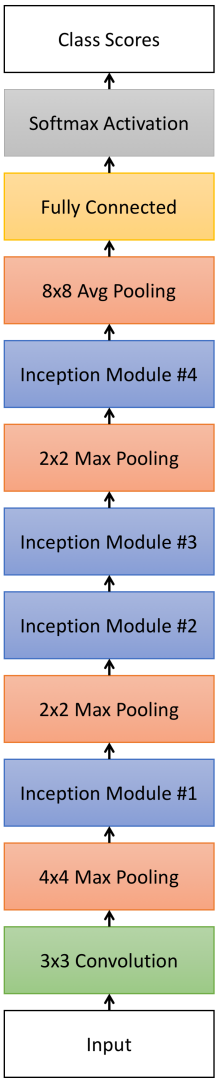

Figure 2: High level overview of the used CNN architecture

The input to our CNN is a 64×64 pixels image with 3 channels from the RGB color representation. It is fed into a convolutional layer with subsequent batch normalization and ReLU activation. A 4×4 maximum pooling layer reduces the spatial dimensions before the first inception module. After a 2×2 maximum pooling layer there are two more inception modules. The last inception module follows another 2×2 maximum pooling layer. In order to reduce the number of parameters introduced by fully connected layers,

we placed a 8×8 average pooling layer before the final affine layer. Finally, a softmax function is used to compute the class losses.

Tables 1 to 4 in the appendix provide additional information regarding the filter sizes and numbers for the layers in the CNNs and in the inception modules, respectively. In tables 2 and 4, the *red* columns as well as the *pool* column denote the 1×1 convolutional layers used for dimensionality reduction.

While the baseline architecture is basically our first guess, the refined CNN is the result of experimenting with different parameters in order to obtain a CNN, which generalizes and does well on the validation set.

Our Python implementation was inspired from the PyTorch implementation of the GoogleNet [2].

2.3. Training

2.3.1 Initial Hyperparameter Tuning

Before seriously training our Convolutional Neural Network, we were interested in finding some suitable values for the learning rate and the weight decay hyperparameters. To do so, we trained the baseline CNN with 53 randomly selected configurations for eight epochs and compared the final validation accuracies. This hyperparameter testing was performed on a reduced size dataset to keep the computational expenses low; both training and validation set were roughly 10% of the original data set size.

The results of this investigation are presented in Figure 3. A learning rate of $1e{-}4$ and a weight decay of $3.16e{-}5$ were chosen for the longer training of the CNNs.

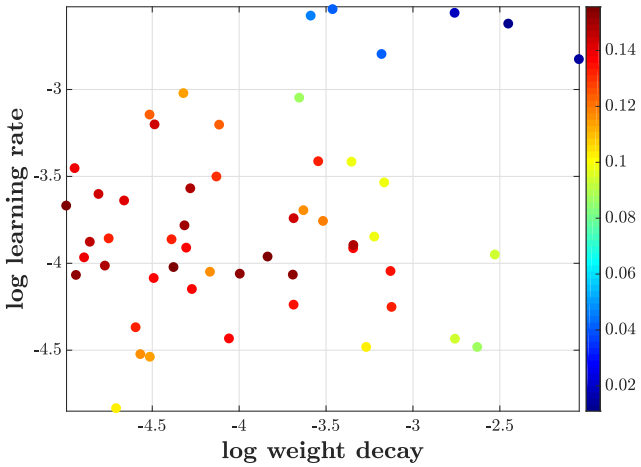

Figure 3: Scatter plot of validation accuracy for hyperparameter search on small dataset after 8 epochs of training

2.3.2 Training

The convolutional neural networks were trained with these hyperparameters for 40 epochs. In addition, a smaller-scale hyperparameter testing was done on the more advanced CNN and the original data set in order to investigate possible differences due to the alteration of the architecture and the different training set size. The results confirmed the choice of the hyperparameters. The training of the baseline CNN was done using the original dataset without the gray scale images, the more advanced CNN was trained on the augmented dataset. This second CNN also uses dropout regularization to combat overfitting, and after 25 epochs the learning rate was manually reduced.

3. Results and Discussion

Figure 4 shows the plotted accuracy histories for both CNNs. The baseline CNN reaches validation accuracies of $40 - 45\%$; however, it is clearly overfitting. The second presented CNN performs similarly well on the validation set, but it does significantly better in terms of generalizing. It turned out to be quite difficult to improve the CNN with respect to both objectives, and we did not improve much further than 45% validation accuracy.

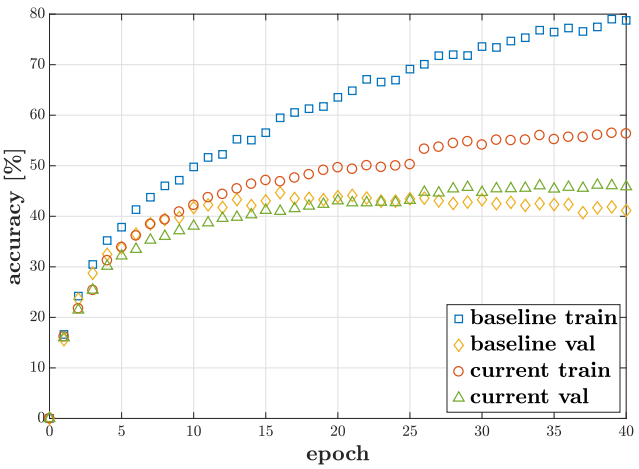

Figure 4: Accuracy history plot

The test accuracies of the baseline and the more advanced Convolutional Neural Network were 37.0% and 41.3%, respectively.

3.1. Dimensionality Reduction

Before the fully connected layer in the advanced CNN, the image is represented by 320 different features. We were interested in whether we could visualize how the different classes were separated at that stage. The visualization of a 320 dimensional space is practically not doable, and so we

applied two different dimensionality reduction techniques to get down to 3 dimensions.

The first technique is Principal Component Analysis (PCA), presented in figure 5, and the second one is t-distributed stochastic neighbor embedding (t-SNE), presented in figure 6. In both figures, the samples of the first three classes from the validation dataset are shown, labeled as 0, 1, and 2 in the legend.

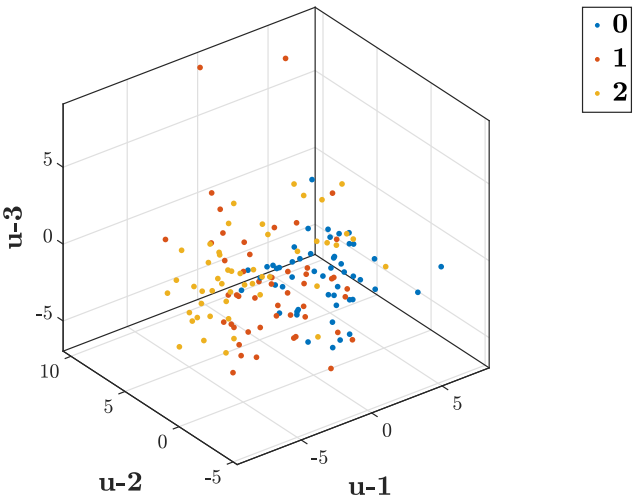

Figure 5: PCA

The reduced representation from the PCA captures only 15.97% of the total variance in the dataset. It is not yet able to well separate the different classes. With t-SNE, the separation works better: Apart from some outliers the classes are each clustered in separate regions.

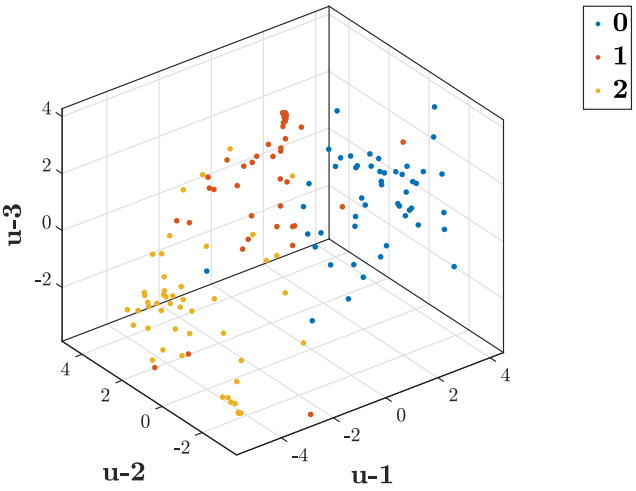

Figure 6: t-SNE

These results generally match our expectations. While we acknowledge that the reduction from 320 to 3 dimensions must be accompanied by a substantial loss of information and will not yield a perfect separation, both techniques to some degree clustered the different classes in separate areas of the 3 dimensional space.

3.2. Feature Visualization

When training a Convolutional Neural Network or when using a trained one, direct diagnosis is limited to discussing the input images and the output class scores. Everything happening in between is not as easy to interpret and therefore less commonly looked at. One milestone paper in this area of research came from Zeiler and Fergus [7], in which they presented some techniques to visualize intermediate layers and discussed how it could influence our understanding of CNNs, motivating the present discussion.

In order get some insight on how to improve our architecture, we explored two different ways of visualizing what is happening in the middle layers. The first method uses backpropagation of the activation of one neuron on the original image pixels as inspired from Erhan et al. [3]. The second method, from Zeiler and Fergus [7], uses a so-called Deconvolutional Network to reconstruct the pixel map of the original image that led to the considered activation of a neuron.

In both cases, we chose neurons from our inception layers and looked for the 9 images from the validation set that activated the most one particular neuron with respect to a specific image norm. We compared both the infinity norm and the L2 norm to get a quantification of how much a neuron is activated by the input image. For the backpropagation, the results were visually more insightful when using the infinity norm compared to the L2 norm. The opposite was found for the Deconvolution results. We then build a map of the pixels from the original image related to the activation of the neuron using either one of the two methods. The output had the dimensions of the input images, and the values were squashed into the RGB range so that we could visualize it. Some of our results are gathered in annex B.

3.2.1 Backpropagation

In the case of the backpropagation, we computed the gradients of the activation map of the neuron with respect to the pixels of the original image. This yielded a quantification of how sensitive the neuron is to each original pixel. The intuition behind this method is that interesting features present an important gradient with respect to the activation and that backpropagation yields the typical pattern that would activate the neuron.

Figure 7 presents results for a 3×3 convolutional layer from the second inception module. The labels are indicated as numbers: the ones located on top of the gradient images are the ones predicted by our network, whereas the ones located at the bottom of the original images are the true label

values. One can see how the neuron seems to be sensitive to the edges of the objects. Such edge detection neurons are found to appear rather frequently in the early layers.

One general observation was that when using this method for a filter from an early layer, the top 9 images did not as often have the same class labels but shared other characteristics such as a dominant color in the image. In contrast, the top 9 images mainly had the same labels for filters from later layers. Indeed, Figures 9 and 10 feature activation for blobs of yellow and green colors, whereas Figures 15 and 16 present a 'flag neuron' and a 'penguin neuron'. These considerations agree with our expectations that the filters learned at later layers should be more complex and more specific to different classes, so that the top 9 images from such a filter predominantly come from one common class.

Second, qualitative comparisons highlighted how some neurons do not feature any specific activation pattern. This could shed some light on how one could improve the architecture adjusting the number of different layers.

In general, the backpropagation images were not as insightful as we had hoped for as they looked relatively similar across all visualized filters. Still, it was possible to recognize characteristic elements from the original images in the backpropagated visualizations, especially for the earlier layers.

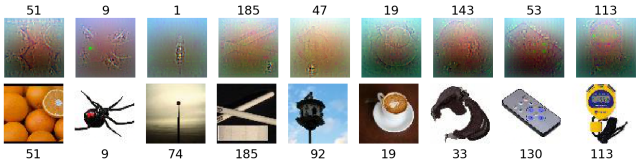

Figure 7: Visualization of one 3×3 convolution neuron from the second inception module (backpropagation)

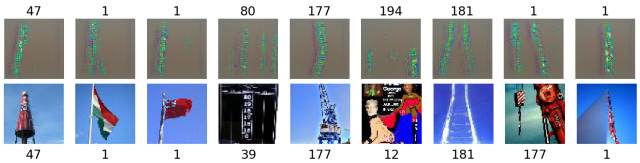

Figure 8: Visualization of one 5×5 convolution neuron from the second inception module (deconvolution)

3.2.2 Deconvolution

The second method for feature visualization relied on the usage of so-called Deconvolutional Neural Networks. Originated in [7], Deconvolution refers to building a network that reverses the operations of the original one in a backward pass. This paper presented highly insightful results enabling further diagnosis to understand each neuron's action. Due to the irreversible nature of some operations of the studied CNN, the Deconvolution Network is not a strict mathematical inverse, but it aims at reconstructing the original image from the feature representation at some neuron in the CNN. This is to say it tries to rebuild the original map of pixels that led to the specific activation featured, instead of the map of pixels that are most sensitive to a variation in the activation, as does backpropagation.

We implemented a Deconvolutional Network according to the same main guidelines as presented in the paper. Inversion of convolutional layer were performed using convolutional layer with transposed weights, ReLU were used as inverse of ReLU, and unpooling layers of max pooling layers were built using the forward pass maximum indices. We removed the batch normalization in the backward pass as doing so highly improved the quality of our results.

Figure 8 presents results for a 5×5 convolutional layer from the second inception module. The neuron seems to be sensitive to vertical pole shapes, as occuring on flag or crane images. The quality of the images is globally improved compared to the backpropagation: the background is rather uniform, and the activated features appear more distinctively.

Like for the backpropagation approach, it was confirmed that early layers were more receptive to simple features, such as oriented edges (vertical poles on Figure 17, horizontal stripes on Figure 18), whereas deeper layers showed higher activation for more abstracts concepts (cups for Figure 23 and cars for Figure 24).

One important observation was that the second and third inception module visualizations did not enable such clear qualitative distinctions. One could have expected the 'car' neuron from Figure 24 to be the result of the activation of some 'wheel' and 'windshield' neurons from these inner layers, but we were not able to highlight such behavior. However, due to these considerations, it is worth noting that we also tried a similar architecture with only three inception modules. We were able to retrieve a similar performance which suggests that we were not able to train our network in such a way that a deeper architecture would clearly outperform a simpler one. One solution to go further would have been to use transfer learning and see if we could have reached better results with our deeper architecture.

4. Conclusion

The original architecture of the GoogleNet was designed for a 1000 label classification dataset. We inspired ourselves from this architecture to see how inception modules scale for a reduced size classification problem with 200 labels. Our architecture involved four successive inception modules and was trained using a dataset of $198,179$ images obtained through some data augmentation techniques. We were able to reduce overfitting using dropout without loss of performance for a validation accuracy around 45%.

Test accuracy was found to be 41.3%. Further improvement of the generalization and validation accuracy became more difficult. To better understand the limitations we performed some visualizations of our trained CNN using methods developed from the literature. The obtained results were not as insightful as examples from state-of-the-art papers, but still provided us with some intuition on what could be improved. Several considerations from our visualization results made us wonder whether we were properly training the intermediate layers of our deep architecture. One interesting step to further this study would be to use transfer learning to see if one could capitalize on a deeper architecture and go beyond 45% validation accuracy.

5. Acknowledgements

The convolutional neural networks were built using the PyTorch deep learning framework. All computations were done using the Google Cloud Computing Platform.

A. Architecture details

Type	Filter size / stride	Output size
convolution	3×3 / 1	$62\times62\times128$
max pool	4×4 / 2	$30\times30\times128$
inception #1		$30\times30\times128$
max pool	2×2 / 2	$16\times16\times128$
inception #2		$16\times16\times128$
inception #3		$16\times16\times256$
max pool	2×2 / 2	$8\times8\times256$
inception #4		$8\times8\times256$
avg pool	8×8 / 1	$1\times1\times256$
fully connected		$1\times1\times200$
softmax		$1\times1\times200$

Table 1: Baseline CNN architecture

#	1×1	3×3 red	3×3	5×5 red	5×5	pool
1	32	48	64	8	16	16
2	32	48	64	8	16	16
3	64	96	128	16	32	32
4	64	96	128	16	32	32

Table 2: Baseline inception module filter numbers

Type	Filter size / stride	Output size
convolution	3×3 / 1	$62\times62\times128$
max pool	4×4 / 2	$30\times30\times128$
inception #1		$30\times30\times128$
max pool	2×2 / 2	$16\times16\times128$
inception #2		$16\times16\times128$
inception #3		$16\times16\times256$
max pool	2×2 / 2	$8\times8\times256$
inception #4		$8\times8\times320$
avg pool	8×8 / 1	$1\times1\times320$
fully connected		$1\times1\times200$
softmax		$1\times1\times200$

Table 3: Current CNN architecture

#	1×1	3×3 red	3×3	5×5 red	5×5	pool
1	32	48	64	8	16	16
2	32	48	64	8	16	16
3	64	96	128	16	32	32
4	80	120	160	20	40	40

Table 4: Current inception module filter numbers

B. Visualization Images

In this annex are presented some visualization results from section 3.2. The labels are indicated as numbers: the top labels are the ones predicted by the network, whereas the bottom ones are the actual ones.

B.1. Backpropagation

B.1.1 First Inception Module

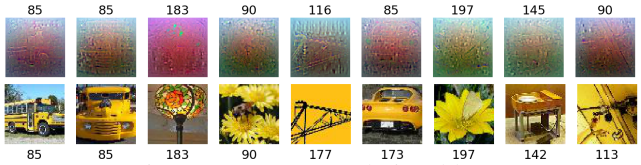

Figure 9: 1×1 convolutional layer

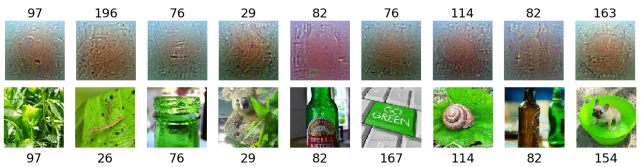

Figure 10: Pool layer

B.1.2 Second Inception Module

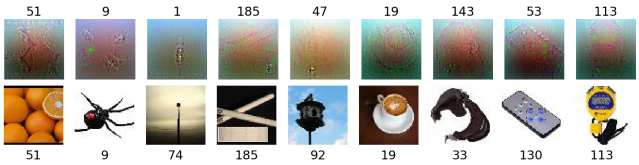

Figure 11: 3×3 convolutional layer

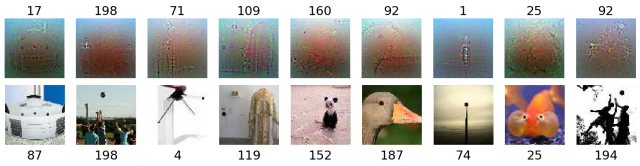

Figure 12: 3×3 convolutional layer

B.1.3 Third Inception Module

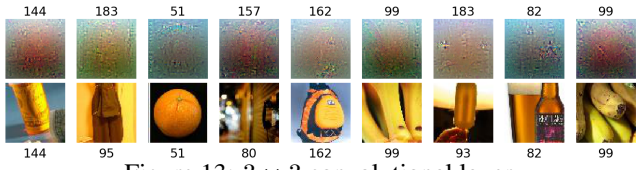

Figure 13: 3×3 convolutional layer

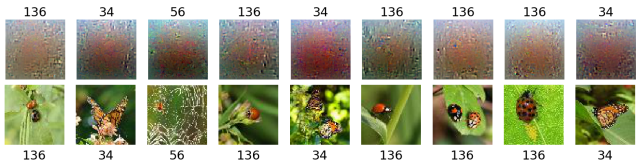

Figure 14: Pool layer

B.1.4 Fourth Inception Module

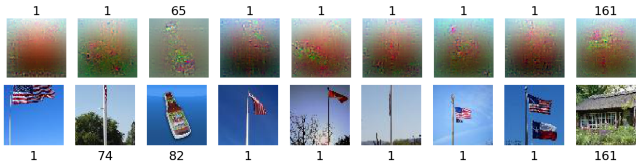

Figure 15: 3×3 convolutional layer

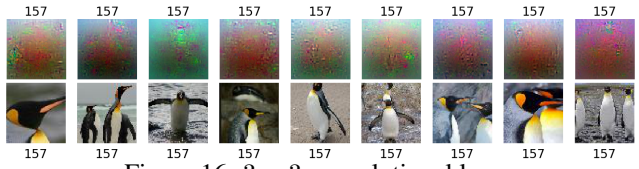

Figure 16: 3×3 convolutional layer

B.2. Deconvolution

B.2.1 First Inception Module

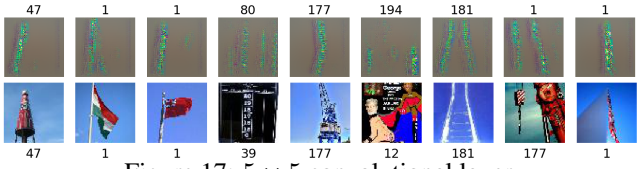

Figure 17: 5×5 convolutional layer

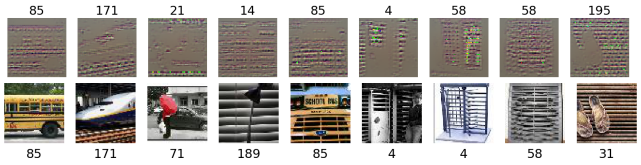

Figure 18: 1×1 convolutional layer

B.2.2 Second Inception Module

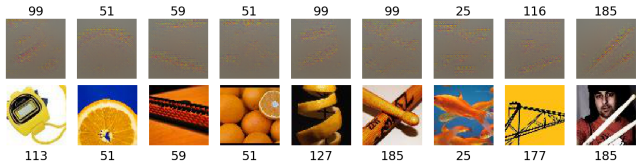

Figure 19: 3×3 convolutional layer

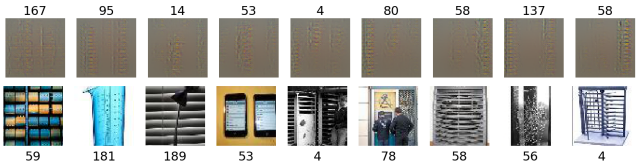

Figure 20: 3×3 convolutional layer

B.2.3 Third Inception Module

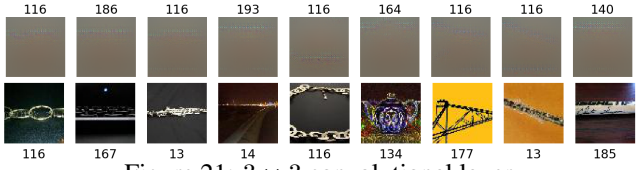

Figure 21: 3×3 convolutional layer

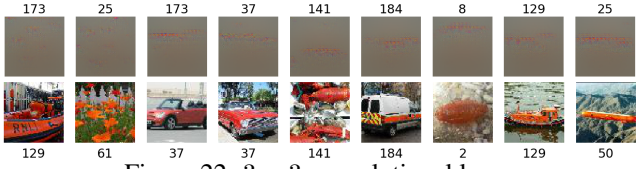

Figure 22: 3×3 convolutional layer

B.2.4 Fourth Inception Module

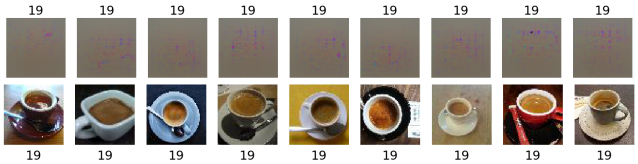

Figure 23: 3×3 convolutional layer

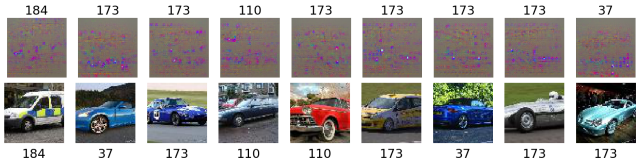

Figure 24: 5×5 convolutional layer

References

[1] ImageNet website. `http://www.image-net.org/`. Accessed: 2017-05-16.

[2] Pytorch googlenet implementation. `https://github.com/pytorch/vision/tree/master/torchvision/models`. Accessed: 2017-05-16.

[3] D. Erhan, Y. Bengio, A. Courville, and P. Vincent. Visualizing higher-layer features of a deep network. Technical Report 1341, University of Montreal, June 2009.

[4] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[5] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[6] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[7] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. *CoRR*, abs/1311.2901, 2013.