Going Broader and Deeper with Convolutions

Abhijit Balaji
axb190013@utdallas.edu
Dec 31, 2012

Abstract

In this paper we introduce a new network architecture called as DeepNet [1] (This is a modification of Inception-Resnet in 2020 timeframe) which achieves the state of the art in the image classification task of CIFAR-10 dataset. In this paper we introduce 2 new design paradigms which form the backbone of DeepNet. These are Residual connections and Inception style blocks. To enable fast, smooth and better convergence we introduce a new weight initialization technique called as Xavier Initialization. We also introduce a new adaptive update rule to the vanilla SGD which we call as Adam. We also introduce another regularization technique called as Dropout which ensures better generalization of the network.

1 Introduction

Image classification is a process in computer vision which aims to classify the dominant object in the image. This is a hard task because, computers see these images as a grid of pixels. On the first look it may seem like we can hand engineer feature extractors to extract relevant information from the image which should result in a good enough classifier. But it is an arduous task because most natural images have challenges like object occlusion, poor lighting, rotated objects, different scales of objects etc.

Traditionally, image classification is done using carefully hand engineered feature selections and use an SVM [2] in the end. With the emergence of AlexNet [3] this trend is seeming to shift. In this paper we have taken inspiration from the above and created a new network architecture called as DeepNet (This a modification of Inception-Resnet [1] in 2020) which achieves state of the art in image classification on CIFAR-10 [4] dataset.

The paper is organized as follows. First, we see a few related works. Then we move on to the design section where we present Residual connections and Inception style design paradigms which motivates the design of DeepNet. Then we move on to the training section where we introduce a few tricks like Batch Normalization, Xavier Initialization and Dropout which ensures stable and fast training. Then we provide our results on CIFAR-10 dataset and further provide the compute performance analysis of our network in the implementation section.

2 Related Works

2.1 Design

Convolutional neural networks (CNNs) have recently enjoyed great success in large scale image recognition challenges. LeNet [5] was one of the first architectures which uses CNNs to classify images. This architecture consists of 2 sets of conv + tanh + average pooling stacks and then a conv layer which is further connected to two fully connected layers and are trained using Stochastic Gradient Descent. It took a decade to understand the magnanimity of this work because back in 1998 we didn't have access to hardware accelerators like GPUs. CNNs became popular when AlexNet [3] was introduced in 2012. The network had a very similar architecture to LeNet, but was deeper, bigger, and featured Convolutional Layers stacked on top of each other. It used ReLU non-linearities instead of Tanh. There are few problems with these architectures. First, flattening the feature maps and connecting fully connected layers increases the number of parameters in the network exponentially. Secondly, blindly increasing the number of layers to increase the expressiveness of the network will result in training collapse because of vanishing gradients. And the initialization of the network plays a very crucial role in the rate of its convergence.

2.2 Training

Minibatch Stochastic Gradient Descent is the dominant method used to train deep neural networks. It works by having the model make predictions on training data and using the error on the predictions to update the model in such a way as to reduce the error. Mini-batch gradient descent is a variation of the gradient descent algorithm that splits the training dataset into small batches that are used to calculate model error and update model coefficients. Mini-batch gradient descent seeks to find a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent. The minibatch SGD has a few limitations. Since most of the neural networks are not strictly convex, there is very high possibility that the vanilla SGD gets stuck in a local optimum. There is no way the vanilla SGD can come out of a local optimum because, the gradient of the loss function at the local optimum is 0. Mini batch requires the configuration of an additional "batch size" hyperparameter for the learning algorithm. The Error information must be accumulated across batches.

2.3 Implementation

The most important operation in neural networks is the matrix multiplication operation. This is also the bottle neck operation. The matrix - matrix multiplication has an Arithmetic intensity of N/3. This means that we can be more efficient as N increases. x86 CPU architectures are designed more to do more sequential tasks. Thus, training these neural networks on CPUs is not ideal. GPUs on the other hand are designed for doing parallel computing. This is because a typical GPU has 1000s of cores compared to a CPU which has like 4-10 cores. The fundamental building block of a CNN is the convolution operation. We can see that this convolution operation can be reduced into 2D matrix-matrix multiplication operations. Thus, Convolution operations will enjoy around 50 times more speed when done on a GPU as compared to when done on a CPU. Thus, it is ideal to implement these CNNs on a GPU.

3 Design

3.1 Motivation and Design Paradigms

The most straightforward way of improving the learning capability of a network is to increase its size. This includes increasing its depth and its width. Blindly increasing the depth of the models results in vanishing gradients. Secondly, there has been various debates on the size of the convolutional filters to use. Whether to use a 11 x 11 filter or a 7 x 7 filter or a 5 x 5 filter etc. Also, while going deep, in the process we need to make sure that the number of parameters in the model do not explode. We tackle the problem of vanishing gradients by using Residual Connections [6] and we tackle the other two issues using an architecture style which we call as Inception [7].

3.2 Residual Connection

Let's consider a function $H(x)$ which is learned by the network (a stack of layers). If we can learn the above function, then without loss of generality we can assume that we can also learn a function $F(x) = H(x) - x$. Thus, we can achieve the same result as the function $H(x)$ by $F(x) + x$. This helps in reducing the vanishing gradients problem because during backpropagation, the add operation always takes the gradient of its output and distributes it equally to all of its inputs, regardless of what their values were during the forward pass. Thus, we will always have non vanishing gradients in the shortcut connection irrespective of whether the other branch's gradient becomes 0 due to repeated application of chain rule. This residual connection is shown in Figure 1. Our experiments show that using a scaling factor α between 0 and 1 such that $H(x) = F(x) + \alpha x$ results in a stable training.

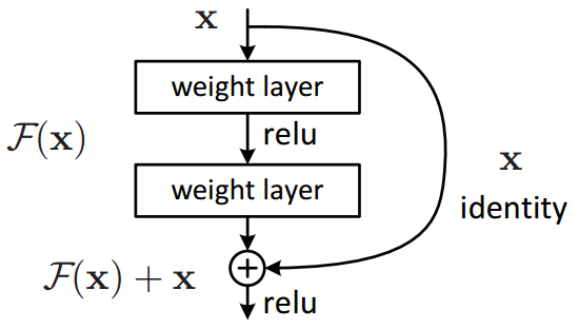

Figure 1: Residual connection. Image Source [6]

3.3 Inception style architecture.

One important hyperparameter of the model is deciding upon the size of the convolutional filter. In inception style we decided to use all of these in parallel. And concatenate them depth wise. The only requirement is that we should make sure that all the branches should do the same spatial reduction (if at all they do any) of the

input features from the previous layer. Another non-obvious realization is that we can replace a 5x5 convolution filter (25 parameters excluding the bias and the number of channels) by a stack of two 3x3 convolutional filters (18 parameters excluding the bias and the number of channels). This is because a stack of two 3x3 filters have the same field of view of a single 5x5. By doing this we have only $\frac{18}{25} \sim 72\%$ of the original number of parameters. One more thing which did to reduce the number of parameters in the network is by using 1x1 filters. Here we reduce the number of parameters by reducing the number of filters in the 1x1 convolution operation. These are shown in Figure 2 and Figure 3.

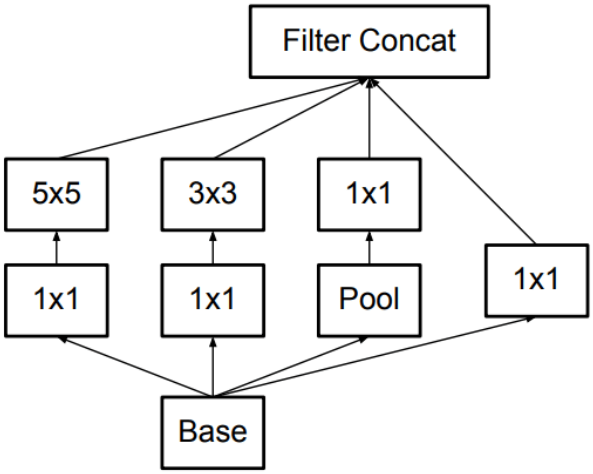

Figure 2: Inception style architecture with 1x1 filters. Image source: [8]

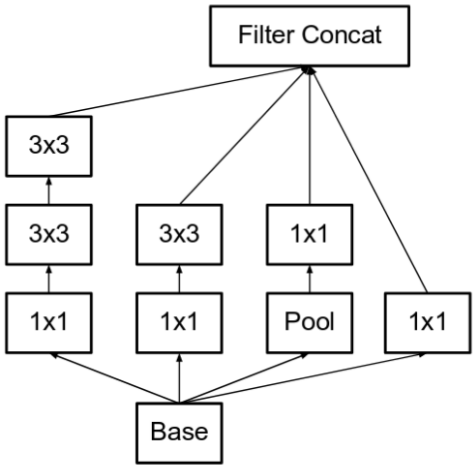

Figure 3: An equivalent inception style architecture to Figure 2 where the 5x5 filter is replaced by a stack of two 3x3 filters resulting in a 38% reduction in parameters. Image source [8]

3.4 Architecture

In this section we describe how we use the above two different design paradigms to create our network which we call as "DeepNet" (This is actually Inception-Resnet or Inception-V4 [1]). We have 2 different types of blocks. The first block is called as an inception block shown in Figure 4 and the second block is called as inception resnet block shown in Figure 5. These two blocks are alternated to create DeepNet. The architecture details are given in Table 1. The network has a total of 1,082,538 parameters. We also used a scaling factor between 0 and 1 while creating the Residual connection.

Input Image (28x28x3)
32 x (3x3) convolutions
1 x Inception block without spatial dimension reduction
2 x Inception Resnet block without spatial dimension reduction and final activation = "ReLU". Scale = 0.17
1 x Inception block with spatial reduction by 2
2 x Inception Resnet block without spatial dimension reduction and final activation = "ReLU". Scale = 0.1
1 x Inception block with spatial reduction by 2
2 x Inception Resnet block without spatial dimension reduction and final activation = "ReLU". Scale = 0.2
1 x Inception Resnet block without final activation. Scale = 1.0
512 x (1x1) convolutions
Global Average Pooling
Fully Connected layer with 10 outputs (CIFAR) for predictions

Table 1: DeepNet Architecture

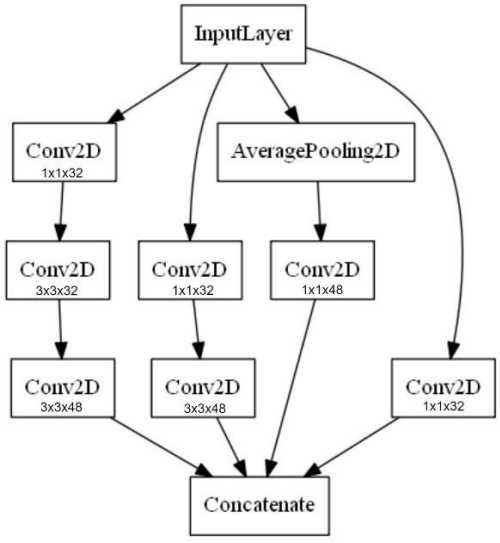

Figure 4: Inception block

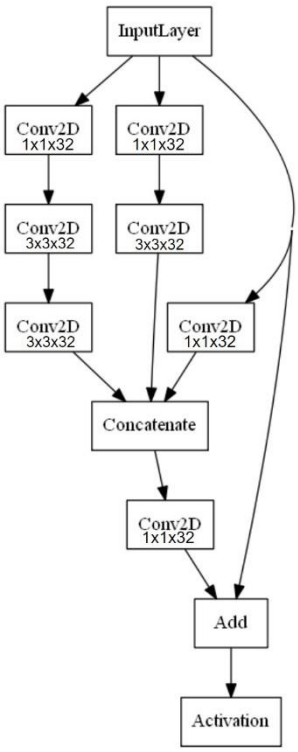

Figure 5: Inception with Resnet block

4 Training

Training a Deep Neural network is a challenging task. And it is more challenging to train a network that is arbitrarily deep. So, in order to make the training process a lot smoother we introduce a few techniques. First, we introduce a weight initialization strategy [9] which initializes the layer's weights to values chosen from a random uniform distribution that's bounded by Equation 1. This weight initialization would maintain the variance of activations and back-propagated gradients all the way up or down the layers of the network.

$$\pm \frac{\sqrt{6}}{\sqrt{fan\_in} + \sqrt{fan\_out}}$$

Equation 1

Another strategy which we used to make the training much easier is called Batch Normalization [10]. Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization and makes it notoriously hard to train models with saturating nonlinearities. Batch Norm impacts network training in a fundamental way: it makes the landscape of the corresponding optimization problem be significantly smoother. This ensures, that the gradients are more predictive and thus allow for use of larger range of learning rates and faster network convergence. The goal of Batch Normalization is to achieve a stable distribution of activation values throughout training, and in our experiments, we apply it before the nonlinearity since that is where matching the first and second moments is more likely to result in a stable distribution. Batch normalization equation is given in Equation 2. In all the convolution layers mentioned in section 3.4 Architecture we use Conv + BN + ReLU.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Equation 2

Additionally, we propose a variant to the SGD update rule which has similar positive effects of a second order update method. We call this method Adam [11]. We do the following change to the SGD update:

- Compute the gradient and its element-wise square using the current parameters.
- Update the exponential moving average of the 1st-order moment and the 2nd-order moment.
- Compute an unbiased average of the 1st-order moment and 2nd-order moment.
- Compute weight update: 1st-order moment unbiased average divided by the square root of 2nd-order moment unbiased average (and scale by learning rate).
- Apply update to the weights.

We also used another regularization technique called as Dropout [12] to improve the generalization capability of the network. By dropping a unit out, we mean temporarily removing it from the network, along with all its incoming and outgoing connections.

We trained DeepNet on CIFAR-10 [4] dataset and the results are presented in Table 2. The loss and accuracy plots are given in Figure 6

S. No	Top1 Error	Top1 Accuracy	Notes
1	7.53%	92.47%	Without Dropout in the end
2	6.85%	93.15%	With 0.3 Dropout in the end

Table 2: Training results on CIFAR-10

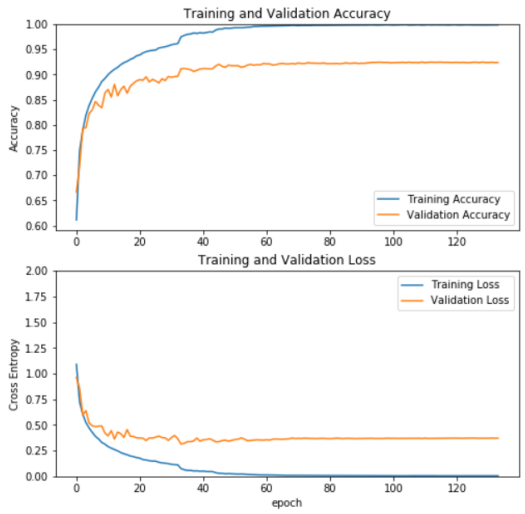

Figure 6: accuracy and loss plots

5 Implementation

We implemented the network using a framework called Tensorflow [13]. We trained the network using a single GTX 1060 with 6GB GPU memory and on a system with 16GB RAM memory. Upon profiling the training code, we see that most of the time is spent on the compute part. This shows that the data loading part is efficient. If this is not the case, then we would have seen a lot of empty space in GPU compute. We also see that the Memory copy from destination to host and vice versa is efficient. Thus, our training code is optimized with most of the time being spent in the hardware accelerator (GPU). The profiler output is given in Figure 7. We also show other on-device time of various other Tensorflow Operations. This is given in Figure 9. From these we can conclude that our training script is not I/O bound as it spends more time in the compute part.

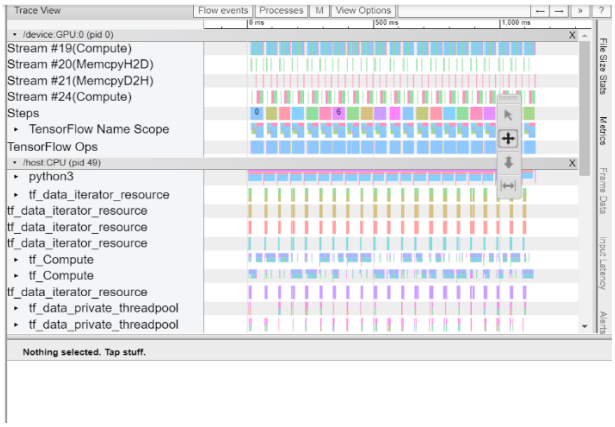

Figure 7: Profiler output

We also looked at the top 10 operations which took the most time on the GPU. The backpropagation through the Conv2D is the bottleneck operation consuming 1.6% of the time. It can be seen in Table 3.

Time (%)	Cumulative Time (%)	Category	Operation
1.6%	1.6%	Conv2DBackpropInput	gradient_tape/InceptionResNet/conv2d_20/Conv2DBackpropInput
1.2%	2.8%	Conv2DBackpropFilter	gradient_tape/InceptionResNet/block35_1_conv/Conv2DBackpropFilter
1.2%	4%	Conv2DBackpropFilter	gradient_tape/InceptionResNet/block35_0_conv/Conv2DBackpropFilter
1%	5.1%	FusedBatchNormGradV3	gradient_tape/InceptionResNet/block35_0_conv_bn/FusedBatchNormGradV3
1%	6.1%	FusedBatchNormGradV3	gradient_tape/InceptionResNet/block35_1_conv_bn/FusedBatchNormGradV3
0.9%	7%	AddN	AddN_7
0.9%	7.9%	AddN	AddN_8
0.9%	8.8%	Conv2DBackpropFilter	gradient_tape/InceptionResNet/conv2d_6/Conv2DBackpropFilter
0.9%	9.7%	Conv2DBackpropFilter	gradient_tape/InceptionResNet/conv2d_3/Conv2DBackpropFilter
0.9%	10.6%	Conv2DBackpropFilter	gradient_tape/InceptionResNet/conv2d_19/Conv2DBackpropFilter

Table 3: Top10 Tensorflow operations on GPU

Since Tensorflow runs GPU kernels in the backend. It is also worthwhile to see the Top10 GPU kernels. This is given in Figure 8.

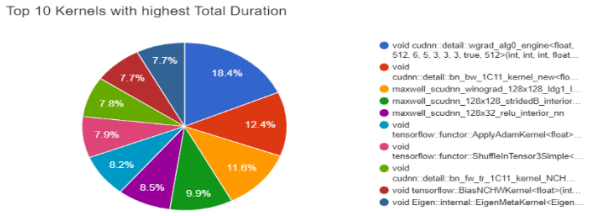

Figure 8: Top 10 GPU Kernels with highest total duration

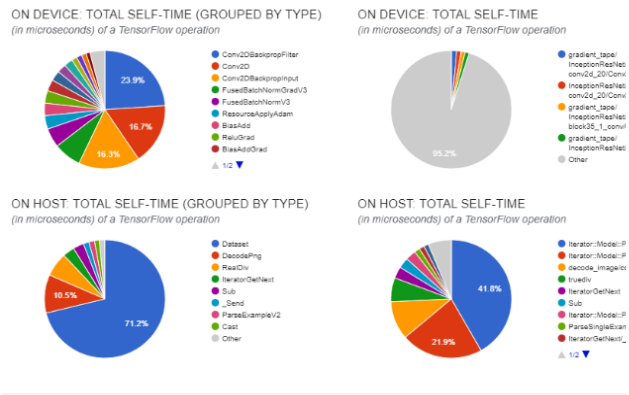

Figure 9

Further we provide the prediction capabilities of DeepNet in a hypothetical hardware. The details are given in Appendix section which follows after the Reference section.

6 Conclusion

In this paper we introduced our new architecture which we call as DeepNet which is based on our two new design paradigms: Residual connections and Inception. These two ensure that we can train arbitrarily deep networks without having vanishing gradients. We also introduced other techniques such as Batch Normalization, Xavier Initialization and Drop out which helps in stable and fast convergence. Our proposed network performed considerably better than other known networks on CIFAR-10 and achieve the state of the art!

Bibliography

[1] C. S. a. S. I. a. V. V. a. A. Alemi, "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning," *arXiv e-print,* no. 1602.07261, 2016.

[2] M. A. Hearst, "Support Vector Machines," *IEEE Intelligent Systems,* 1998.

[3] A. . Krizhevsky, I. . Sutskever and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Communications of the ACM,* vol. 60, no. 6, p. 84–90, .

[4] A. Krizhevsky, *Learning multiple layers of features from tiny images,* 2009.

[5] L. B. Y. B. a. P. H. Y. Lecun, "Gradient-based learning applied to document recognition," *IEEE,* 1998.

[6] X. Z. S. R. a. J. S. K. He, "Deep Residual Learning for Image Recognition," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[7] C. S. e. al, "Going deeper with convolutions," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Boston, 2015.

[8] V. V. S. I. J. S. a. Z. W. C. Szegedy, "Rethinking the Inception Architecture for Computer Vision," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, 2016.

[9] Y. B. Xavier Glorot, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of Machine Learning Research*, 2010.

[10] S. a. S. C. Ioffe, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, 2015.

[11] D. P. K. a. J. Ba, "Adam: A Method for Stochastic Optimization," *CoRR,* 2014.

[12] G. H. A. K. I. S. R. S. Nitish Srivastava, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," in *Journal of Machine Learning Research*, 2014.

[13] A. A. P. B. E. B. Z. C. C. C. Mart´ın Abadi, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," Google, 2015.

Appendix

Here we predict the performance of our model in a hypothetical hardware that consists of an infinite external memory, 1 GB/s DDR bus for data movement between external memory and internal memory, 1 MB of internal memory, 1 TFLOPS of matrix compute and 10 GFLOPS of generic host compute. The total prediction time is data movement time + compute time = 0.0105796199 seconds.

S.No	layer_name	parameters	size_in_bytes	size_in_gb	data_movement_time in_secs	compute_time in_secs
1	conv2d	896	7168	7.17E-06	7.17E-06	1.79E-06
2	batch_normalization	96	768	7.68E-07	7.68E-07	1.92E-07
4	conv2d_4	1056	8448	8.45E-06	8.45E-06	2.11E-06
5	batch_normalization_4	96	768	7.68E-07	7.68E-07	1.92E-07
7	conv2d_2	1056	8448	8.45E-06	8.45E-06	2.11E-06
8	conv2d_5	9248	73984	7.40E-05	7.40E-05	1.85E-05
9	batch_normalization_2	96	768	7.68E-07	7.68E-07	1.92E-07
10	batch_normalization_5	96	768	7.68E-07	7.68E-07	1.92E-07
14	conv2d_1	1056	8448	8.45E-06	8.45E-06	2.11E-06
15	conv2d_3	13872	110976	0.000110976	0.000110976	2.77E-05
16	conv2d_6	13872	110976	0.000110976	0.000110976	2.77E-05
17	conv2d_7	1584	12672	1.27E-05	1.27E-05	3.17E-06
18	batch_normalization_1	96	768	7.68E-07	7.68E-07	1.92E-07
19	batch_normalization_3	144	1152	1.15E-06	1.15E-06	2.88E-07
20	batch_normalization_6	144	1152	1.15E-06	1.15E-06	2.88E-07
21	batch_normalization_7	144	1152	1.15E-06	1.15E-06	2.88E-07
27	conv2d_11	5664	45312	4.53E-05	4.53E-05	1.13E-05
28	batch_normalization_11	96	768	7.68E-07	7.68E-07	1.92E-07
30	conv2d_9	5664	45312	4.53E-05	4.53E-05	1.13E-05
31	conv2d_12	9248	73984	7.40E-05	7.40E-05	1.85E-05
32	batch_normalization_9	96	768	7.68E-07	7.68E-07	1.92E-07
33	batch_normalization_12	96	768	7.68E-07	7.68E-07	1.92E-07
36	conv2d_8	5664	45312	4.53E-05	4.53E-05	1.13E-05
37	conv2d_10	9248	73984	7.40E-05	7.40E-05	1.85E-05
38	conv2d_13	13872	110976	0.000110976	0.000110976	2.77E-05
39	batch_normalization_8	96	768	7.68E-07	7.68E-07	1.92E-07
40	batch_normalization_10	96	768	7.68E-07	7.68E-07	1.92E-07
41	batch_normalization_13	144	1152	1.15E-06	1.15E-06	2.88E-07
46	block35_0_conv	19888	159104	0.000159104	0.000159104	3.98E-05
47	block35_0_conv_bn	528	4224	4.22E-06	4.22E-06	1.06E-06
50	conv2d_17	5664	45312	4.53E-05	4.53E-05	1.13E-05
51	batch_normalization_17	96	768	7.68E-07	7.68E-07	1.92E-07

53	conv2d_15	5664	45312	4.53E-05	4.53E-05	1.13E-05
54	conv2d_18	9248	73984	7.40E-05	7.40E-05	1.85E-05
55	batch_normalization_15	96	768	7.68E-07	7.68E-07	1.92E-07
56	batch_normalization_18	96	768	7.68E-07	7.68E-07	1.92E-07
59	conv2d_14	5664	45312	4.53E-05	4.53E-05	1.13E-05
60	conv2d_16	9248	73984	7.40E-05	7.40E-05	1.85E-05
61	conv2d_19	13872	110976	0.000110976	0.000110976	2.77E-05
62	batch_normalization_14	96	768	7.68E-07	7.68E-07	1.92E-07
63	batch_normalization_16	96	768	7.68E-07	7.68E-07	1.92E-07
64	batch_normalization_19	144	1152	1.15E-06	1.15E-06	2.88E-07
69	block35_1_conv	19888	159104	0.000159104	0.000159104	3.98E-05
70	block35_1_conv_bn	528	4224	4.22E-06	4.22E-06	1.06E-06
73	conv2d_21	5664	45312	4.53E-05	4.53E-05	1.13E-05
74	batch_normalization_21	96	768	7.68E-07	7.68E-07	1.92E-07
76	conv2d_22	9248	73984	7.40E-05	7.40E-05	1.85E-05
77	batch_normalization_22	96	768	7.68E-07	7.68E-07	1.92E-07
79	conv2d_20	50720	405760	0.00040576	0.00040576	0.00010144
80	conv2d_23	13872	110976	0.000110976	0.000110976	2.77E-05
81	batch_normalization_20	96	768	7.68E-07	7.68E-07	1.92E-07
82	batch_normalization_23	144	1152	1.15E-06	1.15E-06	2.88E-07
87	conv2d_27	8224	65792	6.58E-05	6.58E-05	1.64E-05
88	batch_normalization_27	96	768	7.68E-07	7.68E-07	1.92E-07
90	conv2d_25	8224	65792	6.58E-05	6.58E-05	1.64E-05
91	conv2d_28	9248	73984	7.40E-05	7.40E-05	1.85E-05
92	batch_normalization_25	96	768	7.68E-07	7.68E-07	1.92E-07
93	batch_normalization_28	96	768	7.68E-07	7.68E-07	1.92E-07
96	conv2d_24	8224	65792	6.58E-05	6.58E-05	1.64E-05
97	conv2d_26	9248	73984	7.40E-05	7.40E-05	1.85E-05
98	conv2d_29	13872	110976	0.000110976	0.000110976	2.77E-05
99	batch_normalization_24	96	768	7.68E-07	7.68E-07	1.92E-07
100	batch_normalization_26	96	768	7.68E-07	7.68E-07	1.92E-07
101	batch_normalization_29	144	1152	1.15E-06	1.15E-06	2.88E-07
106	block35_2_conv	28928	231424	0.000231424	0.000231424	5.79E-05
107	block35_2_conv_bn	768	6144	6.14E-06	6.14E-06	1.54E-06
110	conv2d_33	8224	65792	6.58E-05	6.58E-05	1.64E-05
111	batch_normalization_33	96	768	7.68E-07	7.68E-07	1.92E-07
113	conv2d_31	8224	65792	6.58E-05	6.58E-05	1.64E-05

114	conv2d_34	9248	73984	7.40E-05	7.40E-05	1.85E-05
115	batch_normalization_31	96	768	7.68E-07	7.68E-07	1.92E-07
116	batch_normalization_34	96	768	7.68E-07	7.68E-07	1.92E-07
119	conv2d_30	8224	65792	6.58E-05	6.58E-05	1.64E-05
120	conv2d_32	9248	73984	7.40E-05	7.40E-05	1.85E-05
121	conv2d_35	13872	110976	0.000110976	0.000110976	2.77E-05
122	batch_normalization_30	96	768	7.68E-07	7.68E-07	1.92E-07
123	batch_normalization_32	96	768	7.68E-07	7.68E-07	1.92E-07
124	batch_normalization_35	144	1152	1.15E-06	1.15E-06	2.88E-07
129	block35_3_conv	28928	231424	0.000231424	0.000231424	5.79E-05
130	block35_3_conv_bn	768	6144	6.14E-06	6.14E-06	1.54E-06
133	conv2d_40	8224	65792	6.58E-05	6.58E-05	1.64E-05
134	batch_normalization_40	96	768	7.68E-07	7.68E-07	1.92E-07
136	conv2d_36	8224	65792	6.58E-05	6.58E-05	1.64E-05
137	conv2d_38	8224	65792	6.58E-05	6.58E-05	1.64E-05
138	conv2d_41	9248	73984	7.40E-05	7.40E-05	1.85E-05
139	batch_normalization_36	96	768	7.68E-07	7.68E-07	1.92E-07
140	batch_normalization_38	96	768	7.68E-07	7.68E-07	1.92E-07
141	batch_normalization_41	96	768	7.68E-07	7.68E-07	1.92E-07
145	conv2d_37	9248	73984	7.40E-05	7.40E-05	1.85E-05
146	conv2d_39	13872	110976	0.000110976	0.000110976	2.77E-05
147	conv2d_42	13872	110976	0.000110976	0.000110976	2.77E-05
148	batch_normalization_37	96	768	7.68E-07	7.68E-07	1.92E-07
149	batch_normalization_39	144	1152	1.15E-06	1.15E-06	2.88E-07
150	batch_normalization_42	144	1152	1.15E-06	1.15E-06	2.88E-07
156	conv2d_46	12320	98560	9.86E-05	9.86E-05	2.46E-05
157	batch_normalization_46	96	768	7.68E-07	7.68E-07	1.92E-07
159	conv2d_44	12320	98560	9.86E-05	9.86E-05	2.46E-05
160	conv2d_47	9248	73984	7.40E-05	7.40E-05	1.85E-05
161	batch_normalization_44	96	768	7.68E-07	7.68E-07	1.92E-07
162	batch_normalization_47	96	768	7.68E-07	7.68E-07	1.92E-07
165	conv2d_43	12320	98560	9.86E-05	9.86E-05	2.46E-05
166	conv2d_45	9248	73984	7.40E-05	7.40E-05	1.85E-05
167	conv2d_48	13872	110976	0.000110976	0.000110976	2.77E-05
168	batch_normalization_43	96	768	7.68E-07	7.68E-07	1.92E-07
169	batch_normalization_45	96	768	7.68E-07	7.68E-07	1.92E-07
170	batch_normalization_48	144	1152	1.15E-06	1.15E-06	2.88E-07

175	block35_4_conv	43392	347136	0.000347136	0.000347136	8.68E-05
176	block35_4_conv_bn	1152	9216	9.22E-06	9.22E-06	2.30E-06
179	conv2d_52	12320	98560	9.86E-05	9.86E-05	2.46E-05
180	batch_normalization_52	96	768	7.68E-07	7.68E-07	1.92E-07
182	conv2d_50	12320	98560	9.86E-05	9.86E-05	2.46E-05
183	conv2d_53	9248	73984	7.40E-05	7.40E-05	1.85E-05
184	batch_normalization_50	96	768	7.68E-07	7.68E-07	1.92E-07
185	batch_normalization_53	96	768	7.68E-07	7.68E-07	1.92E-07
188	conv2d_49	12320	98560	9.86E-05	9.86E-05	2.46E-05
189	conv2d_51	9248	73984	7.40E-05	7.40E-05	1.85E-05
190	conv2d_54	13872	110976	0.000110976	0.000110976	2.77E-05
191	batch_normalization_49	96	768	7.68E-07	7.68E-07	1.92E-07
192	batch_normalization_51	96	768	7.68E-07	7.68E-07	1.92E-07
193	batch_normalization_54	144	1152	1.15E-06	1.15E-06	2.88E-07
198	block35_5_conv	43392	347136	0.000347136	0.000347136	8.68E-05
199	block35_5_conv_bn	1152	9216	9.22E-06	9.22E-06	2.30E-06
202	conv2d_58	12320	98560	9.86E-05	9.86E-05	2.46E-05
203	batch_normalization_58	96	768	7.68E-07	7.68E-07	1.92E-07
205	conv2d_56	12320	98560	9.86E-05	9.86E-05	2.46E-05
206	conv2d_59	9248	73984	7.40E-05	7.40E-05	1.85E-05
207	batch_normalization_56	96	768	7.68E-07	7.68E-07	1.92E-07
208	batch_normalization_59	96	768	7.68E-07	7.68E-07	1.92E-07
211	conv2d_55	12320	98560	9.86E-05	9.86E-05	2.46E-05
212	conv2d_57	9248	73984	7.40E-05	7.40E-05	1.85E-05
213	conv2d_60	13872	110976	0.000110976	0.000110976	2.77E-05
214	batch_normalization_55	96	768	7.68E-07	7.68E-07	1.92E-07
215	batch_normalization_57	96	768	7.68E-07	7.68E-07	1.92E-07
216	batch_normalization_60	144	1152	1.15E-06	1.15E-06	2.88E-07
221	block35_6_conv	43392	347136	0.000347136	0.000347136	8.68E-05
222	block35_6_conv_bn	1152	9216	9.22E-06	9.22E-06	2.30E-06
224	conv_7b	197120	1576960	0.00157696	0.00157696	0.00039424
225	conv_7b_bn	1536	12288	1.23E-05	1.23E-05	3.07E-06
228	predictions	5130	41040	4.10E-05	4.10E-05	1.03E-05