Deep Networks for Image Classification

- Mitravind Pattnaik

Introduction

Image Classification is a classical problem in image processing, computer vision and machine learning. Ordering the images in groups and categories based on its properties and features has wide range of real-world applications (for e.g. detecting text from an image, detecting tumours from MRIs or analysing materials for extra-terrestrial life on different planets and their moons etc.). In machine learning, there have been many techniques for solving this problem but in this paper, we will focus on the most successful technique i.e. the deep neural networks.

Neural networks [1] are a set of algorithms, modelled loosely after the human brain, that are designed to recognize patterns. Deep neural networks are neural networks with comparatively way deeper architecture and hence more complexity. With increasing depth in the architecture, the training becomes more difficult as the demand for hardware resources (memory, processing power etc.) increases as well as there are training problems too (like vanishing gradient etc.). As the area of study of deep neural networks is so huge, it has been assigned a its own branch in machine learning namely – Deep Learning [2]. Convolutional neural networks (also known as CNNs) [3] are deep networks which are used to solve image processing and computer vision problems. The main component of these networks, as the name suggests, is convolution operation.

In this paper, we visit the image classification problem from the context of deep learning with two renowned deep networks, VGGNet [4] and GoogLeNet [5]. For experiments, we'll use the MNIST dataset (more about the networks and dataset mentioned above in succeeding sections).

Critical Analysis/ Related Work

As explicit from the above section, we will be looking at two CNNs, i.e. VGGNet and GoogLeNet, in this paper. So, let's begin with a discussion about the main trends in deep learning for image classification since their publication:

1. Increasing the complexity of the architecture for better performance: For smaller datasets (smaller in resolution or shape) like MNIST (which has a shape of 28x28), a good performance can be achieved even with a shallow architecture. But for large datasets like ImageNet (with a shape of 224x224), the CNN needs a deeper architecture to extract considerable number of features for classification. This was not limited to depth but also, to stacking of layers in parallel. The result of research in this were models like ResNet-50 [6], Inception-v1 [7], Inception-v3 [8] etc. which came out as an improvement in terms of architecture depth and complexity. VGGNet was implemented in 4 different architectures with 11, 13, 16, 19 layers. The Inception model was a part GoogLeNet's architecture.

2. Modularity in CNN architecture: Soon it was realized that just going deeper and deeper, without much knowledge of the internal working of the deep network, was not enough. There had to be a smarter way. The research in year 2017 saw a surge for designing generic blocks that can be inserted in any learning stage of a CNN to improve the performance. These generic blocks are modular and can be used to give attention to specific feature-maps of images in the network. This way it is easier to understand the internal working of the network to some extent and use this understanding to improve its performance. Models like Inception-ResNet-V2 [9], ResNeXt-50 [10], GoogLeNet are good examples for this.

Now we will discuss the key ideas that have motivated research on these models:

1. Convolutional Neural Networks: CNNs though were introduced in the late 1980s, the hardware at that time took forever to train even shallower models. It was only after 2006 that it got a revival, and by 2012 NVIDIA had launched its CUDA programming platform and the world saw rise of CNNs. Many architectures were proposed and VGGNet and GoogLeNet were some of it.

2. Use of smaller filters: Filters of sizes 11x11 and 5x5 were replaced with a stack of 3x3 filters layer which provided an additional benefit of low computational complexity by reducing the number of parameters. It was experimentally demonstrated that concurrent placement of small size (3x3) filters could induce the effect of the large size filter (5x5 and 7x7). This idea of concurrent placement of smaller filters was used in both VGGNet and GoogLeNet.

3. Techniques for faster learning: The research for deeper architecture increased after techniques like transfer learning were introduced which made it easier and faster to train models with deeper architectures.

Apart from this, there are some improvements over VGGNet that are worth mentioning –

i. ResNet-50: ResNet-50 consists of 50 layers which is way more not only compared to VGGNet but also to any other network that was proposed at that time. It is one of the early adopters of batch normalization. This way it prevents the saturation and eventual degradation of performance due to too much increase in the depth of the network.

ii. Inception-v1: It is a 22-layer deep model having parallel towers of convolutions with different filters, followed by concatenation, captures different features at 1×1, 3×3 and 5×5, thereby 'clustering' them. Moreover, it deals with the vanishing gradient problem using auxiliary networks.

iii. GoogLeNet: A 22-layers network which combines the concept of inception, batch normalization, auxiliary networks. It adds to the improvement by the combination of these concepts by implementation of point-wise group convolution operation. This way it can achieve performance of deeper networks with comparatively smaller ones. The improvements due to using these techniques in GoogLeNet is evident from the fact that it reduced the number of parameters that VGGNet used from 138 million to 4 million.

Now let's have a look at the limitations of these networks individually –

1. VGGNet –
   Although VGGNet is a well renowned network even for networks ahead of its time (due to its simplicity with the depth and topology), it has one major limitation worth noting. This limitation is that it uses 138 million parameters due to which it is computationally very expensive.

2. GoogLeNet –
   i. One limitation is its topology which is heterogenous, and hence it has to be customized for each individual module.
   ii. Another limitation is that it consists of a bottleneck layer, i.e. the 1x1 convolution layer in mostly all the modules which drastically reduces the number of features and may sometimes result in loss of useful information.

Model Description

In this section, I will discuss the architectures and improvements that I made over the deep networks that I have implemented. I have used MNIST dataset for training and evaluation of the networks.

- Model Architecture

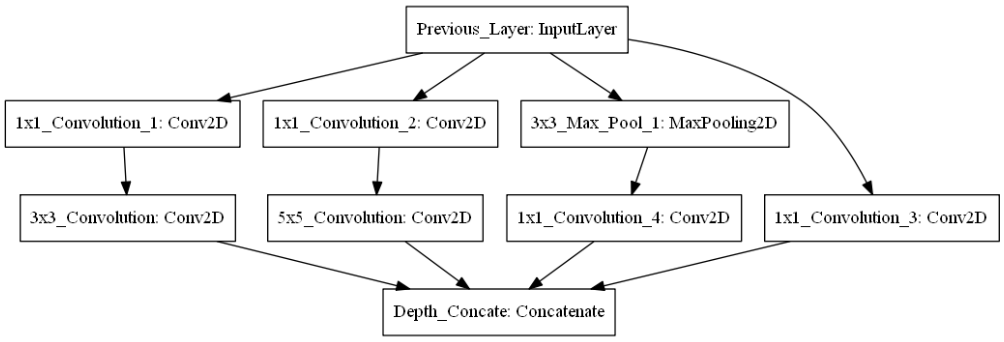
1. VGGNet –

   VGGNet has 4 different architectures, but I have implemented VGG16 that has 16 layers. MNIST is a dataset with images of shape 28x28 pixels for which a network with 16 layers is more than enough considering its small shape and hence I chose this model. The configuration and architecture are provided in the appendix as Table 1 and Figure 1 respectively.

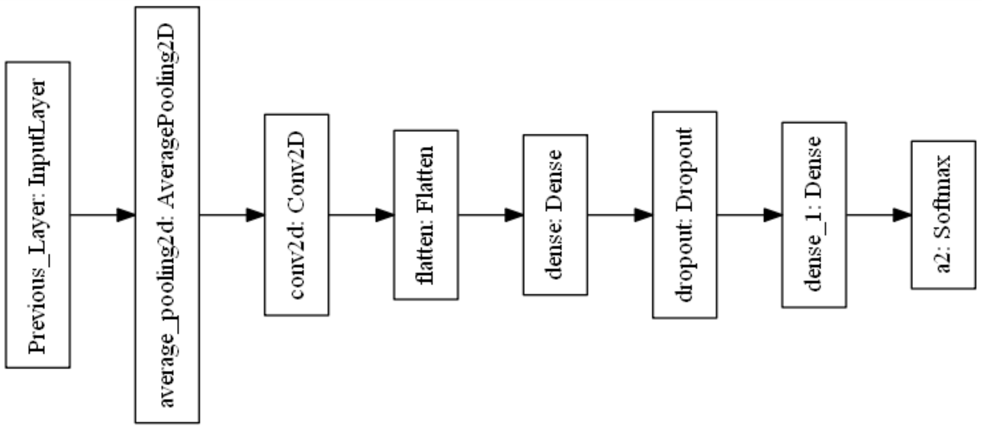
2. GoogLeNet –

   It is a 22 layered architecture with combination of a few techniques as described earlier. Things to note about the architecture are as follows:

   iii. Inception: Inception blocks are optimal local sparse structure of a convolutional vision network can be approximated and covered by readily available dense components. The architecture of inception module implemented is given in the diagram below. The network consists of nine of these inception blocks. Configuration for each of them is provided in the configuration table (in the appendix).


Previous_Layer: InputLayer
1x1_Convolution_1: Conv2D
1x1_Convolution_2: Conv2D
3x3_Max_Pool_1: MaxPooling2D
3x3_Convolution: Conv2D
5x5_Convolution: Conv2D
1x1_Convolution_4: Conv2D
1x1_Convolution_3: Conv2D
Depth_Concate: Concatenate

   iv. Auxiliary layers: These are implemented to propagate the gradients back properly and hence prevent the vanishing gradient problem. The architecture for these layers shown below:



   It has the following configuration:

   - An average pooling layer with 5×5 filter size and stride 3.

- A 1×1 convolution with 128 filters for dimension reduction and rectified linear activation.
- A fully connected layer with 512 units and rectified linear activation.
- A dropout layer with 70% ratio of dropped outputs.
- A linear layer with softmax loss as the classifier

The diagram for architecture of the whole network and configuration table is given in the appendix as Table 2 and Figure 2 respectively.

- Improvements made in the network

  These models are highly optimized in every way and have won prizes in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC). Therefore, I believe that there is not much room for improvement (though not completely denying the scope of improvement completely, there might be scope for improvement).

  I do have managed to make an improvement in the VGGNet, but that is related to the database more than the model itself. One thing that I noted while experiments was that for smaller datasets, decreasing the number of features in the fully connected layers increased the accuracy. Also, this reduced the parameters and therefore, made the training faster. My hypothesis for this is that if the number of features is too big compared to the size of the image, the layers would be more vulnerable to noise and hence reduce the networks performance. Whereas, if the number of features is too less, it will result in loss of data. Therefore, it the number of features has a high dependency to the size of image being used in the dataset.

  In my case, I have used MNIST dataset with the image size 28x28. The experiments done in the VGGNet Paper is done on ImageNet database which has images of size 224x224. Therefore, they have set the number of features to 1024. For my setup, that is too large and hence affects the performance. Therefore, I have set the number of features of the fully connected layers before the output layer to 128, and this gives a better performance than the former case. This can also be seen in the VGGNet configuration table in the appendix.

Experiments

- Datasets:

  I have used only the MNIST dataset for training and evaluation of the networks. The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits. It has a set of 60,000 images for training and 10,000 images for testing. I have further split the training dataset into 50,000 images of training dataset and 10,000 images of validation dataset. An example of a set from MNIST is provided in the appendix as figure 3.

- Experiment Results:

  1. Setup –
     I have used the following configuration as experimental setup for analysis of the models.

     - General Parameters

Epochs	20
Batch Size	100
Optimizer	Adam
Loss	Categorical loss

Metric	Accuracy

- Hyper-parameters –
  i. Activation function
  ii. Learning rate

2. Observation –

We will observe the results of combinations of different learning rates with different activation functions.

- VGGNet –

i. Activation = ReLU.

Learning rate	Loss			Accuracy		
	Training	Validation	Testing	Training	Validation	Testing
0.1	14.55	14.4918	14.5482	0.0968	0.1009	0.0974
0.01	2.3019	2.3019	2.3012	0.1122	0.1064	0.1135
0.001	0.0251	0.0547	0.0409	0.9942	0.9912	0.9931
0.0001	0.0083	0.0306	0.0280	0.9976	0.9936	0.9929
0.000001	0.0283	0.0391	0.0323	0.9910	0.9884	0.9888

ii. Activation = TanH

Learning rate	Loss			Accuracy		
	Training	Validation	Testing	Training	Validation	Testing
0.001	2.3717	2.3684	2.3698	0.1014	0.1090	0.1028
0.0001	0.0126	0.0378	0.0309	0.9959	0.9905	0.9926
0.00001	0.0067	0.0425	0.0375	0.9981	0.9889	0.9893
0.000001	1.0666	1.0173	1.0296	0.7049	0.7210	0.7219

iii. Activation = Leaky ReLU

Learning rate	Loss			Accuracy		
	Training	Validation	Testing	Training	Validation	Testing
0.001	14.5282	14.5208	14.5385	0.0986	0.0991	0.0980
0.0001	0.0094	0.0416	0.0375	0.9975	0.9902	0.9913
0.00001	0.0284	0.0500	0.0409	0.9905	0.9856	0.9874
0.000001	0.2680	0.2398	0.2406	0.9208	0.9278	0.9295
0.0000001	1.7034	1.6379	1.6476	0.4567	0.4900	0.4821

- GoogLeNet –

i. Activation = ReLU

Learning rate	Loss			Accuracy		
	Training	Validation	Testing	Training	Validation	Testing
0.01	6.9133	6.9129	6.9122	0.1033	0.0991	0.0980
0.001	0.0595	0.1424	0.1430	0.9947	0.9899	0.9880
0.0001	0.0296	0.1535	0.0341	0.9969	0.9885	0.9905
0.00001	0.1729	0.1736	0.1500	0.9859	0.9845	0.9851

ii. Activation = TanH

Learning rate	Loss			Accuracy		
	Training	Validation	Testing	Training	Validation	Testing
0.01	10.8347	66.8522	65.7888	0.0976	0.1064	0.1135
0.001	3.1242	2.7873	2.7752	0.9448	0.9489	0.9505
0.0001	0.0298	0.01493	0.1194	0.9962	0.9898	0.9901
0.00001	0.0775	0.1940	0.1686	0.9964	0.9823	0.9834

iii. Activation = Leaky ReLU

Learning rate	Loss			Accuracy		
	Training	Validation	Testing	Training	Validation	Testing
0.01	43.4415	43.4447	43.3657	0.0989	0.0991	0.0980
0.001	17.3203	9.4367	3.5836	0.5401	0.6616	0.6617
0.0001	0.0235	0.1288	0.0869	0.9971	0.9912	0.9919
0.00001	0.1553	0.0579	0.1444	0.9869	0.9829	0.9828

- Further evaluation –
  The experiments show the impact of change in learning rates and activation functions on the performance of VGGNet and GoogLeNet. From the experimental data tabulated above, we can conclude that at specific learning rates, both the models perform almost equally good at any activation function. Among the data obtained, the best result that we got was at the following configuration:

  1. VGGNet
  $$Learning\ rate = 0.01, Activation = ReLU$$
  2. GoogLeNet
  $$Learning\ rate = 0.0001, Activation = Leaky\ ReLU$$

  Therefore, from the data we can conclude that for the MNIST dataset, all the activation functions tested in this experimental setup give excellent performance for their specific learning rates at 20 epochs, a batch size of 100.

Conclusion

In this paper, we compared and evaluated the performance of two distinguished models (VGGNet and GoogLeNet). The evaluation was based on 2 hyper parameters namely- learning rate and activation functions. At least one configuration for each learning rate yielded good results. Both the models can be said to have good performance on MNIST for more than one configuration irrespective of their own limitations.

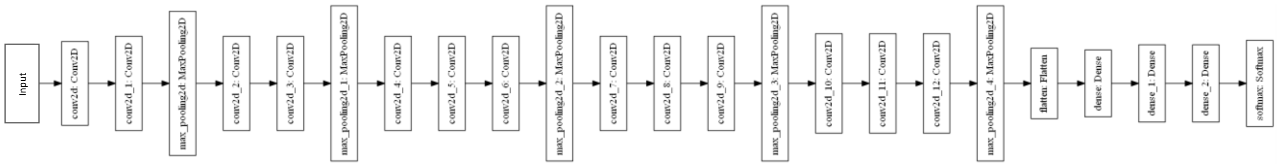
For future work on this, I would like to study the different CNN models including these two and evaluate their performance for different hyper-parameters like epochs, batch size, optimizers etc. Also, would like to study the internal working of these deep networks and come to a better understanding about them. This way I aim to some day design a network which is highly optimized in every way without any limitations (just an ambitious thought).

References

1. Pathmind. *A Beginner's Guide To Neural Networks And Deep Learning*. [online] Available at: https://pathmind.com/wiki/neural-network
2. Investopedia. n.d. *How Deep Learning Can Help Prevent Financial Fraud*. [online] Available at: https://www.investopedia.com/terms/d/deep-learning.asp
3. Medium. *Understanding Of Convolutional Neural Network (CNN) — Deep Learning*. [online] Available at: https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148
4. Simonyan, K. and Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.
5. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A., 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1-9).
6. He, K., Zhang, X., Ren, S. and Sun, J., 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).
7. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A., 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1-9).
8. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J. and Wojna, Z., 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2818-2826).
9. Szegedy, C., Ioffe, S., Vanhoucke, V. and Alemi, A.A., 2017, February. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI conference on artificial intelligence*.
10. Xie, S., Girshick, R., Dollár, P., Tu, Z. and He, K., 2017. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1492-1500).
11. Medium. *Illustrated: 10 CNN Architectures*. [online] Available at: https://towardsdatascience.com/illustrated-10-cnn-architectures-95d78ace614d#81e0

Appendix

- Figure 1

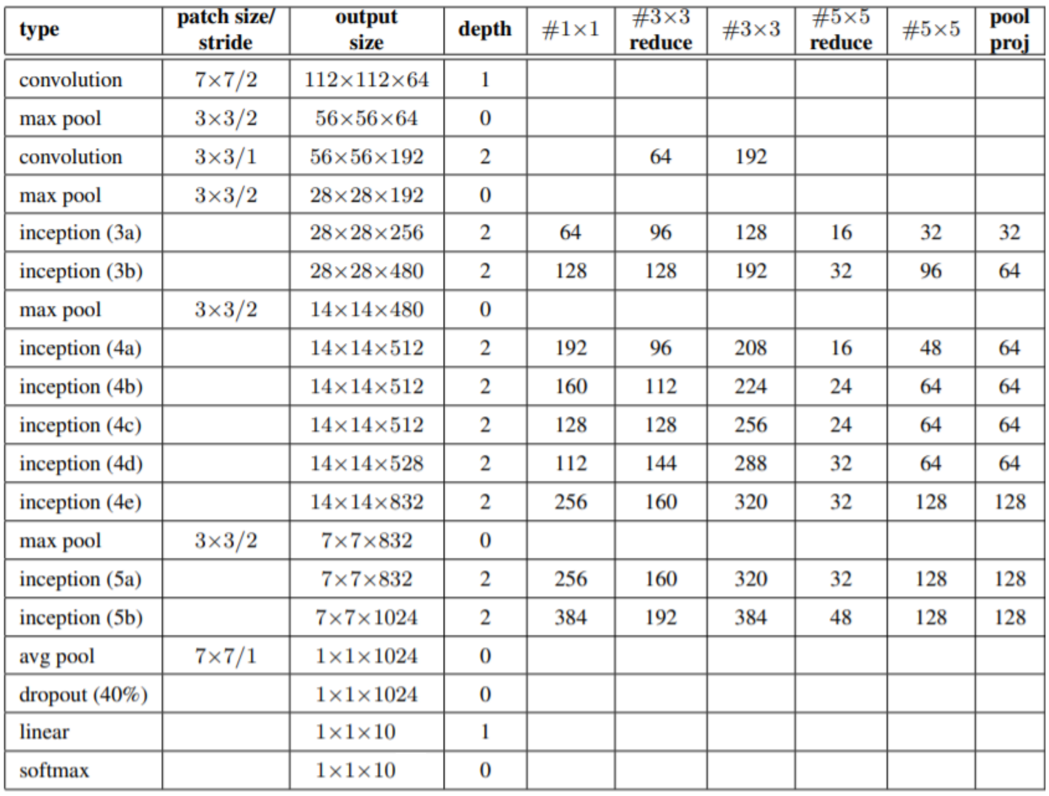


- Table 1

Configuration for the layers sequentially –

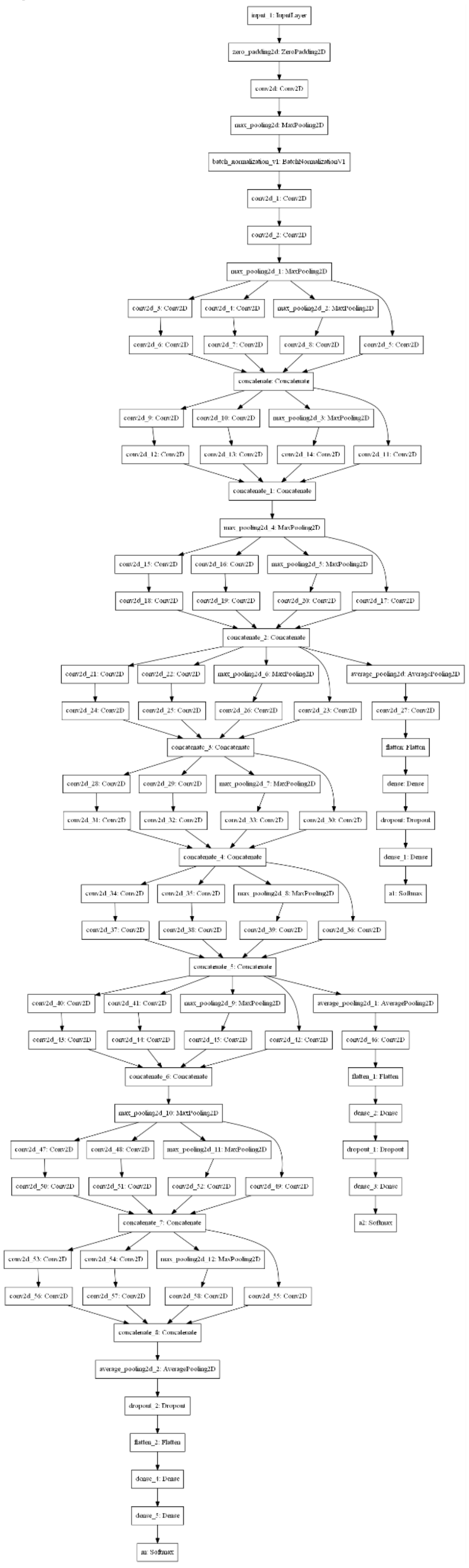
Layer	Config
2 Conv layers	kernel = 3, filter = 64, padding = same
Max Pool	Pool size = 2, stride = 2
2 Conv layers	kernel = 3, filter = 128, padding = same
Max Pool	Pool size = 2, stride = 2
3 Conv layers	kernel = 3, filter = 256, padding = same
Max Pool	Pool size = 2, stride = 2
3 Conv layers	kernel = 3, filter = 512, padding = same
Max Pool	Pool size = 2, stride = 2
3 Conv layers	kernel = 3, filter = 512, padding = same
Max Pool	Pool size = 2, stride = 2
2 Dense layers	Features = 128
Dense layer	Features = 10

- Table 2

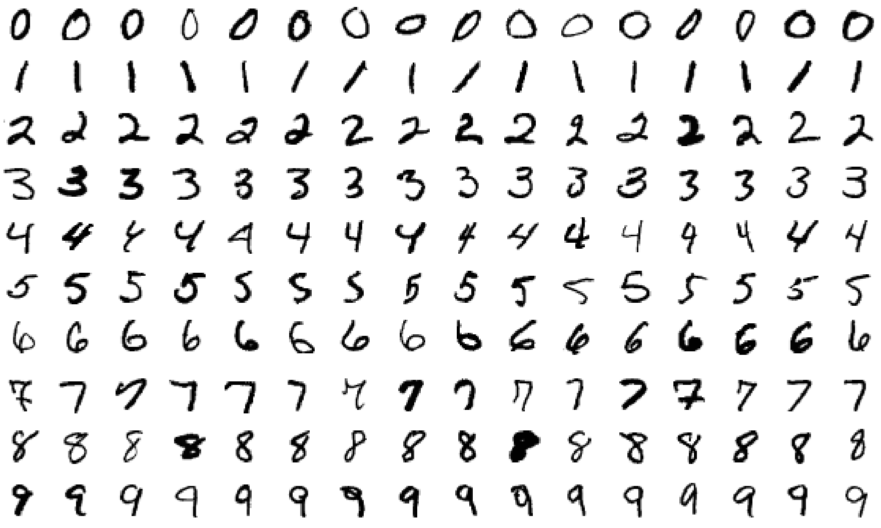


type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj
convolution	7×7/2	112×112×64	1						
max pool	3×3/2	56×56×64	0						
convolution	3×3/1	56×56×192	2		64	192			
max pool	3×3/2	28×28×192	0						
inception (3a)		28×28×256	2	64	96	128	16	32	32
inception (3b)		28×28×480	2	128	128	192	32	96	64
max pool	3×3/2	14×14×480	0						
inception (4a)		14×14×512	2	192	96	208	16	48	64
inception (4b)		14×14×512	2	160	112	224	24	64	64
inception (4c)		14×14×512	2	128	128	256	24	64	64
inception (4d)		14×14×528	2	112	144	288	32	64	64
inception (4e)		14×14×832	2	256	160	320	32	128	128
max pool	3×3/2	7×7×832	0						
inception (5a)		7×7×832	2	256	160	320	32	128	128
inception (5b)		7×7×1024	2	384	192	384	48	128	128
avg pool	7×7/1	1×1×1024	0						
dropout (40%)		1×1×1024	0						
linear		1×1×10	1						
softmax		1×1×10	0						

- Figure 2

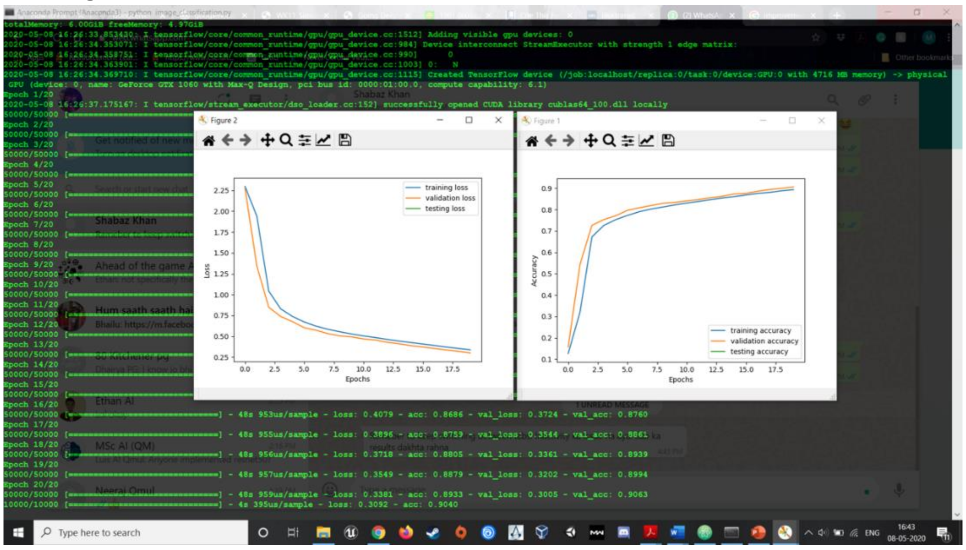

input_1: InputLayer
zero_padding2d: ZeroPadding2D
conv2d: Conv2D
max_pooling2d: MaxPooling2D
batch_normalization_v1: BatchNormalizationV1
conv2d_1: Conv2D
conv2d_2: Conv2D
max_pooling2d_1: MaxPooling2D
conv2d_3: Conv2D
conv2d_4: Conv2D
max_pooling2d_2: MaxPooling2D
conv2d_6: Conv2D
conv2d_7: Conv2D
conv2d_8: Conv2D
conv2d_5: Conv2D
concatenate: Concatenate
conv2d_9: Conv2D
conv2d_10: Conv2D
max_pooling2d_3: MaxPooling2D
conv2d_12: Conv2D
conv2d_13: Conv2D
conv2d_14: Conv2D
conv2d_11: Conv2D
concatenate_1: Concatenate
max_pooling2d_4: MaxPooling2D
conv2d_15: Conv2D
conv2d_16: Conv2D
max_pooling2d_5: MaxPooling2D
conv2d_18: Conv2D
conv2d_19: Conv2D
conv2d_20: Conv2D
conv2d_17: Conv2D
concatenate_2: Concatenate
conv2d_21: Conv2D
conv2d_22: Conv2D
max_pooling2d_6: MaxPooling2D
average_pooling2d: AveragePooling2D
conv2d_24: Conv2D
conv2d_25: Conv2D
conv2d_26: Conv2D
conv2d_23: Conv2D
conv2d_27: Conv2D
concatenate_3: Concatenate
flatten: Flatten
conv2d_28: Conv2D
conv2d_29: Conv2D
max_pooling2d_7: MaxPooling2D
dense: Dense
conv2d_31: Conv2D
conv2d_32: Conv2D
conv2d_33: Conv2D
conv2d_30: Conv2D
dropout: Dropout
concatenate_4: Concatenate
dense_1: Dense
conv2d_34: Conv2D
conv2d_35: Conv2D
max_pooling2d_8: MaxPooling2D
a1: Softmax
conv2d_37: Conv2D
conv2d_38: Conv2D
conv2d_39: Conv2D
conv2d_36: Conv2D
concatenate_5: Concatenate
conv2d_40: Conv2D
conv2d_41: Conv2D
max_pooling2d_9: MaxPooling2D
average_pooling2d_1: AveragePooling2D
conv2d_43: Conv2D
conv2d_44: Conv2D
conv2d_45: Conv2D
conv2d_42: Conv2D
conv2d_46: Conv2D
concatenate_6: Concatenate
flatten_1: Flatten
max_pooling2d_10: MaxPooling2D
dense_2: Dense
conv2d_47: Conv2D
conv2d_48: Conv2D
max_pooling2d_11: MaxPooling2D
dropout_1: Dropout
conv2d_50: Conv2D
conv2d_51: Conv2D
conv2d_52: Conv2D
conv2d_49: Conv2D
dense_3: Dense
concatenate_7: Concatenate
a2: Softmax
conv2d_53: Conv2D
conv2d_54: Conv2D
max_pooling2d_12: MaxPooling2D
conv2d_56: Conv2D
conv2d_57: Conv2D
conv2d_58: Conv2D
conv2d_55: Conv2D
concatenate_8: Concatenate
average_pooling2d_2: AveragePooling2D
dropout_2: Dropout
flatten_2: Flatten
dense_4: Dense
dense_5: Dense
m: Softmax

- Figure 3



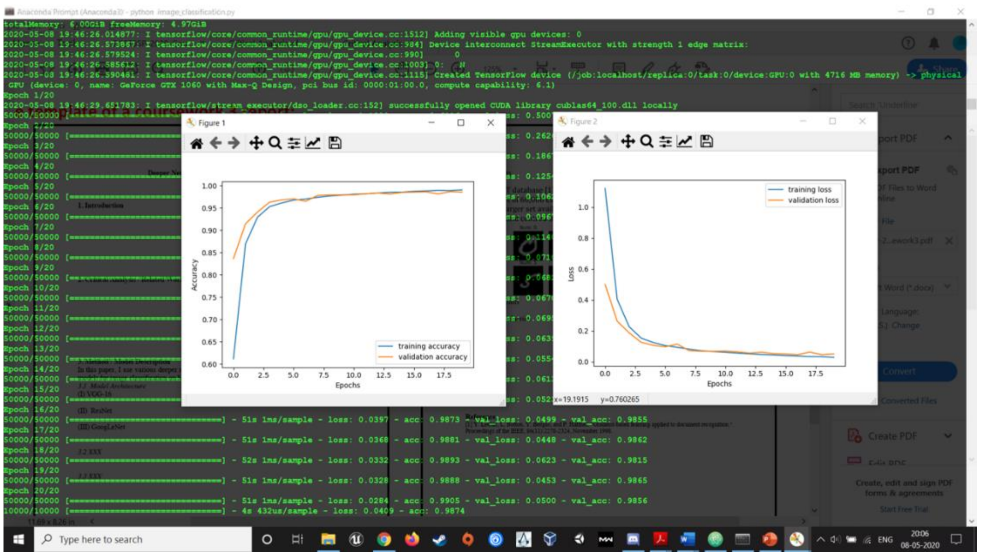
- Below are some run-time screen shots taken during evaluation and experiment phase.

    - Learning rate = 0.00001, Activation = ReLU



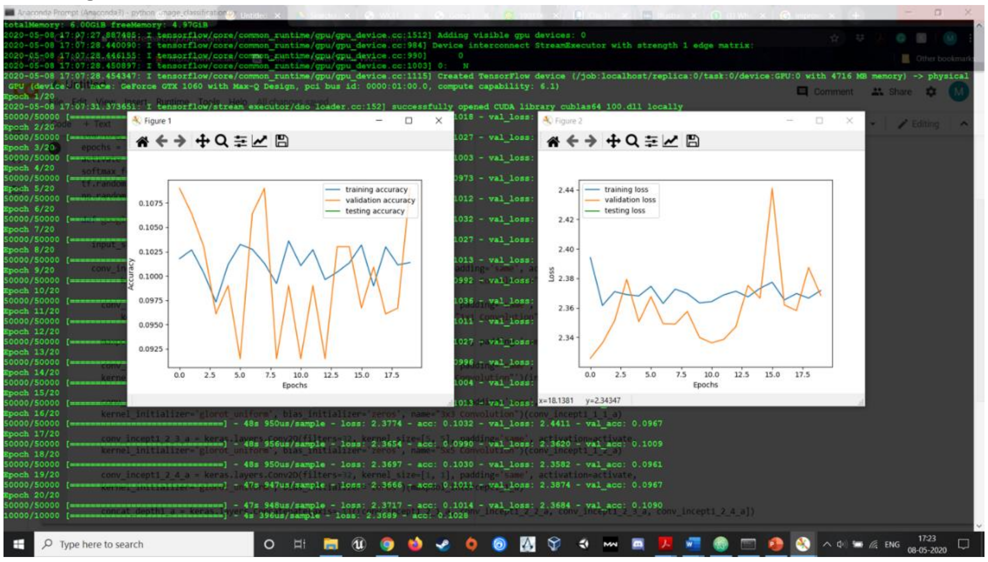
```
totalMemory: 6.00GiB freeMemory: 4.97GiB
2020-05-08 16:26:33.853430: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1512] Adding visible gpu devices: 0
2020-05-08 16:26:34.353071: I tensorflow/core/common_runtime/gpu/gpu_device.cc:984] Device interconnect StreamExecutor with strength 1 edge matrix:
2020-05-08 16:26:34.358751: I tensorflow/core/common_runtime/gpu/gpu_device.cc:990]      0
2020-05-08 16:26:34.363901: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1003] 0:   N
2020-05-08 16:26:34.369710: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1115] Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 4716 MB memory) -> physical
 GPU (device: 0, name: GeForce GTX 1060 with Max-Q Design, pci bus id: 0000:01:00.0, compute capability: 6.1)
Epoch 1/20
2020-05-08 16:26:37.175167: I tensorflow/stream_executor/dso_loader.cc:152] successfully opened CUDA library cublas64_100.dll locally
50000/50000 [==================
Epoch 2/20
50000/50000 [==================
Epoch 3/20
50000/50000 [==================
Epoch 4/20
50000/50000 [==================
Epoch 5/20
50000/50000 [==================
Epoch 6/20
50000/50000 [==================
Epoch 7/20
50000/50000 [==================
Epoch 8/20
50000/50000 [==================
Epoch 9/20
50000/50000 [==================
Epoch 10/20
50000/50000 [==================
Epoch 11/20
50000/50000 [==================
Epoch 12/20
50000/50000 [==================
Epoch 13/20
50000/50000 [==================
Epoch 14/20
50000/50000 [==================
Epoch 15/20
50000/50000 [==================
Epoch 16/20
50000/50000 [==============================] - 48s 953us/sample - loss: 0.4079 - acc: 0.8686 - val_loss: 0.3724 - val_acc: 0.8760
Epoch 17/20
50000/50000 [==============================] - 48s 955us/sample - loss: 0.3896 - acc: 0.8759 - val_loss: 0.3544 - val_acc: 0.8861
Epoch 18/20
50000/50000 [==============================] - 48s 956us/sample - loss: 0.3718 - acc: 0.8805 - val_loss: 0.3361 - val_acc: 0.8939
Epoch 19/20
50000/50000 [==============================] - 48s 957us/sample - loss: 0.3549 - acc: 0.8879 - val_loss: 0.3202 - val_acc: 0.8994
Epoch 20/20
50000/50000 [==============================] - 48s 959us/sample - loss: 0.3381 - acc: 0.8933 - val_loss: 0.3005 - val_acc: 0.9063
10000/10000 [==============================] - 4s 395us/sample - loss: 0.3092 - acc: 0.9040
```

- Learning rate = 0.00001, Activation = Leaky ReLU



```
totalMemory: 6.00GiB freeMemory: 4.97GiB
2020-05-08 19:46:26.014877: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1512] Adding visible gpu devices: 0
2020-05-08 19:46:26.573867: I tensorflow/core/common_runtime/gpu/gpu_device.cc:984] Device interconnect StreamExecutor with strength 1 edge matrix:
2020-05-08 19:46:26.579524: I tensorflow/core/common_runtime/gpu/gpu_device.cc:990]      0
2020-05-08 19:46:26.585612: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1003] 0:   N
2020-05-08 19:46:26.590461: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1115] Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 4716 MB memory) -> physical
 GPU (device: 0, name: GeForce GTX 1060 with Max-Q Design, pci bus id: 0000:01:00.0, compute capability: 6.1)
Epoch 1/20
2020-05-08 19:46:29.651783: I tensorflow/stream_executor/dso_loader.cc:152] successfully opened CUDA library cublas64_100.dll locally
50000/50000 [                                                   ss: 0.500
Epoch 2/20
50000/50000 [=========                                          ss: 0.262
Epoch 3/20
50000/50000 [=========                                          ss: 0.186
Epoch 4/20
50000/50000 [=========                                          ss: 0.125
Epoch 5/20
50000/50000 [=========                                          ss: 0.106
Epoch 6/20
50000/50000 [=========                                          ss: 0.096
Epoch 7/20
50000/50000 [=========                                          ss: 0.114
Epoch 8/20
50000/50000 [=========                                          ss: 0.071
Epoch 9/20
50000/50000 [=========                                          ss: 0.068
Epoch 10/20
50000/50000 [=========                                          ss: 0.067
Epoch 11/20
50000/50000 [=========                                          ss: 0.069
Epoch 12/20
50000/50000 [=========                                          ss: 0.063
Epoch 13/20
50000/50000 [=========                                          ss: 0.055
Epoch 14/20
50000/50000 [=========                                          ss: 0.061
Epoch 15/20
50000/50000 [=========                                          ss: 0.052
Epoch 16/20
50000/50000 [==============================] - 51s 1ms/sample - loss: 0.0397 - acc: 0.9873 - val_loss: 0.0499 - val_acc: 0.9855
Epoch 17/20
50000/50000 [==============================] - 51s 1ms/sample - loss: 0.0368 - acc: 0.9881 - val_loss: 0.0448 - val_acc: 0.9862
Epoch 18/20
50000/50000 [==============================] - 52s 1ms/sample - loss: 0.0332 - acc: 0.9893 - val_loss: 0.0623 - val_acc: 0.9815
Epoch 19/20
50000/50000 [==============================] - 51s 1ms/sample - loss: 0.0328 - acc: 0.9888 - val_loss: 0.0453 - val_acc: 0.9865
Epoch 20/20
50000/50000 [==============================] - 51s 1ms/sample - loss: 0.0284 - acc: 0.9905 - val_loss: 0.0500 - val_acc: 0.9856
10000/10000 [==============================] - 4s 432us/sample - loss: 0.0409 - acc: 0.9874
```

- Learning rate = 0.0001, Activation = TanH



```
totalMemory: 6.00GiB freeMemory: 4.97GiB
2020-05-08 17:07:27.887485: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1512] Adding visible gpu devices: 0
2020-05-08 17:07:28.440090: I tensorflow/core/common_runtime/gpu/gpu_device.cc:984] Device interconnect StreamExecutor with strength 1 edge matrix:
2020-05-08 17:07:28.446155: I tensorflow/core/common_runtime/gpu/gpu_device.cc:990]      0
2020-05-08 17:07:28.450897: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1003] 0:   N
2020-05-08 17:07:28.454347: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1115] Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 4716 MB memory) -> physical
 GPU (device: 0, name: GeForce GTX 1060 with Max-Q Design, pci bus id: 0000:01:00.0, compute capability: 6.1)
Epoch 1/20
2020-05-08 17:07:31.373651: I tensorflow/stream_executor/dso_loader.cc:152] successfully opened CUDA library cublas64_100.dll locally
50000/50000 [========                              1018 - val_loss:
Epoch 2/20
50000/50000 [========                              1027 - val_loss:
Epoch 3/20
50000/50000 [========                              1003 - val_loss:
Epoch 4/20
50000/50000 [========                              0973 - val_loss:
Epoch 5/20
50000/50000 [========                              1012 - val_loss:
Epoch 6/20
50000/50000 [========                              1032 - val_loss:
Epoch 7/20
50000/50000 [========                              1027 - val_loss:
Epoch 8/20
50000/50000 [========                              1013 - val_loss:
Epoch 9/20
50000/50000 [========                              0992 - val_loss:
Epoch 10/20
50000/50000 [========                              1036 - val_loss:
Epoch 11/20
50000/50000 [========                              1011 - val_loss:
Epoch 12/20
50000/50000 [========                              1027 - val_loss:
Epoch 13/20
50000/50000 [========                              0996 - val_loss:
Epoch 14/20
50000/50000 [========                              1004 - val_loss:
Epoch 15/20
50000/50000 [========                              1013 - val_loss:
Epoch 16/20
50000/50000 [==============================] - 48s 950us/sample - loss: 2.3774 - acc: 0.1032 - val_loss: 2.4411 - val_acc: 0.0967
Epoch 17/20
50000/50000 [==============================] - 48s 956us/sample - loss: 2.3654 - acc: 0.0990 - val_loss: 2.3620 - val_acc: 0.1009
Epoch 18/20
50000/50000 [==============================] - 48s 950us/sample - loss: 2.3697 - acc: 0.1030 - val_loss: 2.3582 - val_acc: 0.0961
Epoch 19/20
50000/50000 [==============================] - 47s 947us/sample - loss: 2.3666 - acc: 0.1011 - val_loss: 2.3874 - val_acc: 0.0967
Epoch 20/20
50000/50000 [==============================] - 47s 948us/sample - loss: 2.3717 - acc: 0.1014 - val_loss: 2.3684 - val_acc: 0.1090
10000/10000 [==============================] - 4s 396us/sample - loss: 2.3689 - acc: 0.1028
```