Assignment 2
Computer Vision

Image Classification (CNNs)

Nayera Abd El Samie ***ID:4032***
Esraa Wael El-Hawash ***ID:3959***
Guehad Mohamed ***ID:3861***

Image Classification
Applying CNNs For Image Classification

In this problem set we want to use PyTorch to train convolutional neural network (CNN) based image classifier. You will be using CIFAR-10 dataset to train and evaluate your model. You should try different values for the CNN hyper-parameters and study the effect of each of them on the model performance.

Dataset:

The CIFAR-10 dataset which we'll be using consists of 60,000 32x32 color images in 10 classes, with 6000 images per class. The classes are completely mutually exclusive. There is no overlap between automobiles and trucks. "Automobile" includes sedans, SUVs, things of that sort. "Truck" includes only big trucks. Neither includes pickup trucks.

So the first thing we did was download the data using the keras library, and then we split the data to training data (5000 images) x_train and their labels y_train and then the test data (1000 images) x_test and their labels y_test.

Pre-Processing:

Next step we normalize our dataset images as a form of preprocessing.
We did the normalization manually without the use of any ready functions or library to guarantee a perfect result.

First thing we did was turn the data to type 'float32' we then calculate the mean using the numpy library function np.mean(x_train).
Next we calculate the standard deviation also with the help of numpy np.std(x_train).
Lastly we subtract the mean from the data and divide by the standard deviation and with that the normalization is done.
The same goes for the test data as well.

Another pre-processing procedure is the one hot encoding for the train and test labels.
We use the keras function 'to_categorical' which basically represents the 10 classes as Binary values to match the dimensions in the layers of our CNN.

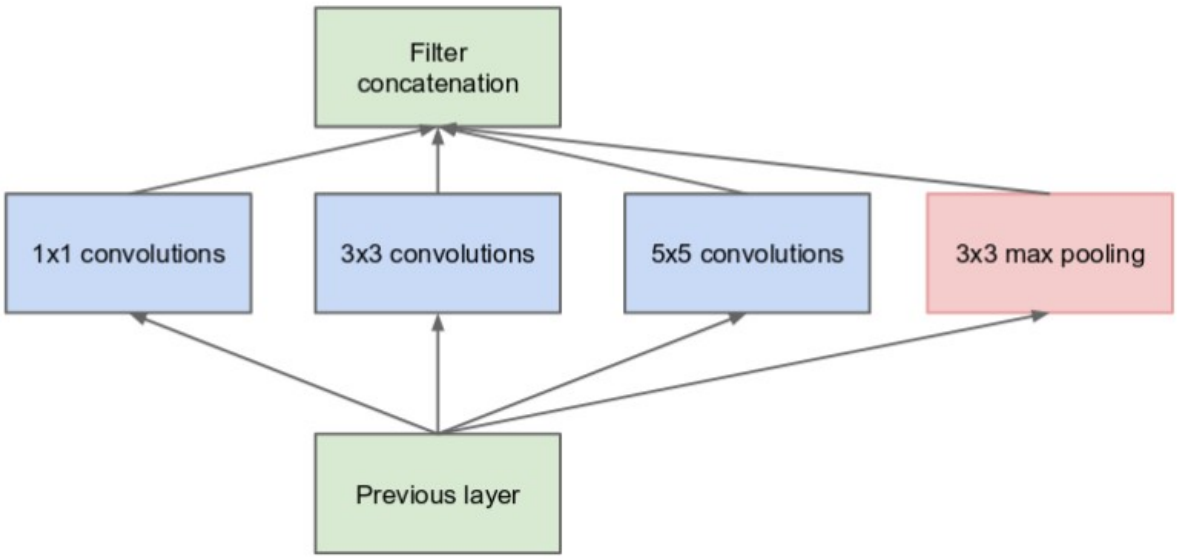
CNN:

Now to the Neural network itself.
Convolutional neural networks (CNNs) are a special kind of neural networks for processing data that has a known grid-like topology such as time-series data and image data. The general structure of a CNN consists of one or more convolutional layers followed by one or more fully connected layers as in a standard multi-layer neural network.

We were required to train the 2 Versions inception modules we studied in lectures and compare their performance.

We implemented the Naive & Dimension Reduction Version 1 of Inception and the Version 2 Dimension reduction.

A) Naive Inception (V1)



(a) Inception module, naïve version

It performs convolution on an input, with 3 different sizes of filters (1x1, 3x3, 5x5). Additionally, max pooling is also performed. The outputs are concatenated and sent to the next inception module.
To build this network we followed the lecture's exact architecture.
We started by initializing the input shape to (32,32,3) 32 x 32 being the image size and 3 is for the RGB channels.

Next up the first convolution layer with 16 filters of size 1x1 with Relu activation function.
Following that another convolution layer with filters of size 3x3 and same activation function.
Next is another conv layer of 3x3 filters and Relu activation function.
Lastly a Maxpool layer with stride 1 .
All these layers are then concatenated together , flattened and then they enter a fully connected neural network using the ‘Dense’ layer.
We added a dropout layer of (0.5) value in between the first and second dense layers to improve our accuracy.
After 3 Dense layers we reach the output layer with 10 neurons for outpupts because of the 10 classes , softmax is the activation function used in the output.
We then compile the model using the adam optimizer and the categorical cross entropy to calculate the loss and the accuracy of the model on the training data.

Fitting the training data has lead to the following results:

1st trial with 3 epochs batch size = 100 and 16 filters

Epoch 1/3 50000/50000 [========] - 58s 1ms/step - loss: 2.3509 - acc: 0.4309

Epoch 2/3 50000/50000 [========] - 56s 1ms/step - loss: 1.1237 - acc: 0.6023

Epoch 3/3 50000/50000 [========] - 56s 1ms/step - loss: 0.9103 - acc: 0.6760

2nd trial with 3 epochs batch size = 200 and 16 filters

Epoch 1/3 50000/50000 [========] - 37s 739us/step - loss: 4.2712 - acc: 0.3582

Epoch 2/3 50000/50000 [========] - 36s 716us/step - loss: 1.2184 - acc: 0.5695

Epoch 3/3 50000/50000 [========] - 36s 713us/step - loss: 0.9614 - acc: 0.6582

108.95607733726501

3rd trial with 3 epochs batch size = 100 and 8 filters

Epoch 1/3 50000/50000 [========] - 34s 671us/step - loss: 2.1968 - acc: 0.4291

Epoch 2/3 50000/50000 [========] - 32s 641us/step - loss: 1.1815 - acc: 0.5800

Epoch 3/3 50000/50000 [========] - 32s 641us/step - loss: 0.9803 - acc: 0.6499

98.24419283866882

4th trial with 10 epochs batch size = 100 and 16 filters

Epoch 1/10 50000/50000 [=======] - 63s 1ms/step - loss: 2.2409 - acc: 0.4383

Epoch 2/10 50000/50000 [=======] - 55s 1ms/step - loss: 1.1261 - acc: 0.6028

Epoch 3/10 50000/50000 [=======] - 55s 1ms/step - loss: 0.9180 - acc: 0.6759

Epoch 4/10 50000/50000 [=======] - 55s 1ms/step - loss: 0.7354 - acc: 0.7411

Epoch 5/10 50000/50000 [=======] - 55s 1ms/step - loss: 0.5865 - acc: 0.7928

Epoch 6/10 50000/50000 [=======] - 55s 1ms/step - loss: 0.4365 - acc: 0.8465

Epoch 7/10 50000/50000 [=======] - 55s 1ms/step - loss: 0.3388 - acc: 0.8800

Epoch 8/10 50000/50000 [=======] - 55s 1ms/step - loss: 0.2568 - acc: 0.9105

Epoch 9/10 50000/50000 [=======] - 55s 1ms/step - loss: 0.1997 - acc: 0.9311

Epoch 10/10 50000/50000 [=======] - 55s 1ms/step - loss: 0.1767 - acc: 0.9390

558.2802133560181

5th trial with 10 epochs batch size = 100 and 8 filters

Epoch 1/10 50000/50000 [=======] - 39s 780us/step - loss: 2.0203 - acc: 0.4216

Epoch 2/10 50000/50000 [=======] - 32s 641us/step - loss: 1.2015 - acc: 0.5719

Epoch 3/10 50000/50000 [=======] - 32s 641us/step - loss: 1.0235 - acc: 0.6352

Epoch 4/10 50000/50000 [=======] - 32s 639us/step - loss: 0.8660 - acc: 0.6899

Epoch 5/10 50000/50000 [=======] - 32s 639us/step - loss: 0.7126 - acc: 0.7463

Epoch 6/10 50000/50000 [=======] - 32s 637us/step - loss: 0.5718 - acc: 0.7955

Epoch 7/10 50000/50000 [=======] - 32s 636us/step - loss: 0.4523 - acc: 0.8395

Epoch 8/10 50000/50000 [=======] - 32s 639us/step - loss: 0.3490 - acc: 0.8764

Epoch 9/10 50000/50000 [=======] - 32s 635us/step - loss: 0.2719 - acc: 0.9048

Epoch 10/10 50000/50000 [=======] - 32s 638us/step - loss: 0.2204 - acc: 0.9230

326.9078528881073

6th trial with 10 epochs batch size = 100, 16 filters and dropout layer = 0.5

Epoch 1/10 50000/50000 [=======] - 58s 1ms/step - loss: 2.0706 - acc: 0.3450

Epoch 2/10 50000/50000 [=======] - 56s 1ms/step - loss: 1.4326 - acc: 0.4831

Epoch 3/10 50000/50000 [=======] - 56s 1ms/step - loss: 1.2523 - acc: 0.5542

Epoch 4/10 50000/50000 [=======] - 56s 1ms/step - loss: 1.1334 - acc: 0.5990

Epoch 5/10 50000/50000 [=======] - 57s 1ms/step - loss: 1.0431 - acc: 0.6311

Epoch 6/10 50000/50000 [=======] - 56s 1ms/step - loss: 0.9577 - acc: 0.6621

Epoch 7/10 50000/50000 [=======] - 56s 1ms/step - loss: 0.8891 - acc: 0.6835

Epoch 8/10 50000/50000 [=======] - 56s 1ms/step - loss: 0.8300 - acc: 0.7067

Epoch 9/10 50000/50000 [=======] - 56s 1ms/step - loss: 0.7544 - acc: 0.7339

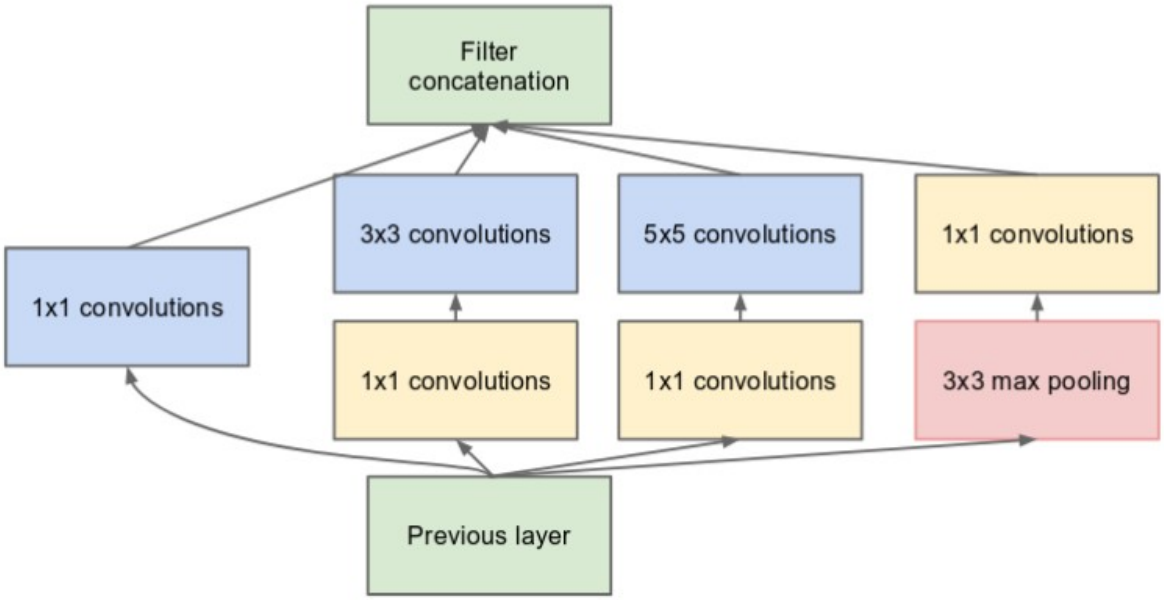
Epoch 10/10 50000/50000 [=======] - 56s 1ms/step - loss: 0.7096 - acc: 0.7506

566.3250277042389

Running on Test:

```
10000/10000 [==============================] - 3s
273us/step
Test accuracy: 0.6637
```

B)Dimension Reduction (V1)



(b) Inception module with dimension reductions

Deep neural networks are computationally expensive. To make it cheaper, the authors limit the number of input channels by adding an extra 1x1 convolution before the 3x3 and 5x5 convolutions. Though adding an extra operation may seem counterintuitive, 1x1 convolutions are far more cheaper than 5x5 convolutions, and the reduced number of input channels also help.
And this is exactly what we did in this dimensionality reduction.
As stated before we start with the same input layer, first layer are 10 1x1 filters, followed by a layer of 1x1 and 3x3 filters, Next a 1x1 and 5x5 filter. Ending with max pooling and another 1x1 layer.
Concatenation is done on all layers, flattening occurs and the same fully connected network is implemented.
The fitting on training data for this network gave the following results:

10 neurons ,epochs 3, batch size 100 :

Epoch 1/3

50000/50000 [==============================] - 58s 1ms/step - loss: 2.3509 - acc: 0.4309

Epoch 2/3

50000/50000 [==============================] - 56s 1ms/step - loss: 1.1237 - acc: 0.6023

Epoch 3/3

50000/50000 [==============================] - 56s 1ms/step - loss: 0.9103 - acc: 0.6760

--

epochs 10, batch size 100 :

Epoch 1/10

50000/50000 [==============================] - 56s 1ms/step - loss: 1.9506 - acc: 0.4289

Epoch 2/10

50000/50000 [==============================] - 48s 963us/step - loss: 1.1588 - acc: 0.5905

Epoch 3/10

50000/50000 [==============================] - 48s 964us/step - loss: 0.8649 - acc: 0.6943

Epoch 4/10

50000/50000 [==============================] - 48s 961us/step - loss: 0.6002 - acc: 0.7879

Epoch 5/10

50000/50000 [==============================] - 48s 967us/step - loss: 0.3593 - acc: 0.8746

Epoch 6/10

50000/50000 [==============================] - 48s 968us/step - loss: 0.1990 - acc: 0.9328

Epoch 7/10

50000/50000 [==============================] - 48s 957us/step - loss: 0.1322 - acc: 0.9553

Epoch 8/10

50000/50000 [==============================] - 48s 962us/step - loss: 0.1039 - acc: 0.9656

Epoch 9/10

50000/50000 [==============================] - 48s 962us/step - loss: 0.0812 - acc: 0.9734

Epoch 10/10

50000/50000 [==============================] - 48s 958us/step - loss: 0.0723 - acc: 0.9765

----

epochs 7, batch size 100 :

Epoch 1/7

50000/50000 [==============================] - 50s 996us/step - loss: 2.0886 - acc: 0.4162

Epoch 2/7

50000/50000 [==============================] - 48s 963us/step - loss: 1.1655 - acc: 0.5838

Epoch 3/7

50000/50000 [==============================] - 48s 965us/step - loss: 0.8862 - acc: 0.6862

Epoch 4/7

50000/50000 [==============================] - 48s 961us/step - loss: 0.6179 - acc: 0.7815

Epoch 5/7

50000/50000 [==============================] - 48s 958us/step - loss: 0.3445 - acc: 0.8787

Epoch 6/7

50000/50000 [==============================] - 48s 957us/step - loss: 0.1775 - acc: 0.9392

Epoch 7/7

50000/50000 [==============================] - 48s 966us/step - loss: 0.1122 - acc: 0.9635

----

epochs 3,batch size 200 :

Epoch 1/3

50000/50000 [==============================] - 33s 669us/step - loss: 2.4864 - acc: 0.3911

Epoch 2/3

50000/50000 [==============================] - 32s 631us/step - loss: 1.2269 - acc: 0.5652

Epoch 3/3

50000/50000 [==============================] - 32s 637us/step - loss: 0.9686 - acc: 0.6569

----

epochs 7,batch size 200 :

Epoch 1/7

50000/50000 [==============================] - 33s 666us/step - loss: 11.7122 - acc: 0.1759

Epoch 2/7

50000/50000 [==============================] - 31s 629us/step - loss: 1.4010 - acc: 0.4948

Epoch 3/7

50000/50000 [==============================] - 31s 627us/step - loss: 1.1414 - acc: 0.5908

Epoch 4/7

50000/50000 [==============================] - 31s 629us/step - loss: 0.9606 - acc: 0.6595

Epoch 5/7

50000/50000 [==============================] - 31s 628us/step - loss: 0.7591 - acc: 0.7324

Epoch 6/7

50000/50000 [==============================] - 31s 629us/step - loss: 0.5217 - acc: 0.8158

Epoch 7/7

50000/50000 [==============================] - 32s 630us/step - loss: 0.3070 - acc: 0.8941

---

epochs 10,batch size 200 :

Epoch 1/10

50000/50000 [==============================] - 34s 672us/step - loss: 2.5419 - acc: 0.3863

Epoch 2/10

50000/50000 [==============================] - 32s 634us/step - loss: 1.2755 - acc: 0.5455

Epoch 3/10

50000/50000 [==============================] - 32s 632us/step - loss: 1.0500 - acc: 0.6274

Epoch 4/10

50000/50000 [==============================] - 32s 631us/step - loss: 0.8271 - acc: 0.7055

Epoch 5/10

50000/50000 [==============================] - 32s 630us/step - loss: 0.6056 - acc: 0.7863

Epoch 6/10

50000/50000 [==============================] - 32s 631us/step - loss: 0.3777 - acc: 0.8685

Epoch 7/10

50000/50000 [==============================] - 32s 631us/step - loss: 0.2022 - acc: 0.9300

Epoch 8/10

50000/50000 [==============================] - 32s 635us/step - loss: 0.1090 - acc: 0.9631

Epoch 9/10

50000/50000 [==============================] - 32s 637us/step - loss: 0.0790 - acc: 0.9741

Epoch 10/10

50000/50000 [==============================] - 32s 638us/step - loss: 0.0582 - acc: 0.9811

----

8 neurons, batch size 100, epochs 10 :

Epoch 1/10

50000/50000 [==============================] - 43s 853us/step - loss: 2.4292 - acc: 0.3974

Epoch 2/10

50000/50000 [==============================] - 40s 807us/step - loss: 1.2274 - acc: 0.5601

Epoch 3/10

50000/50000 [==============================] - 40s 804us/step - loss: 0.9883 - acc: 0.6480

Epoch 4/10

50000/50000 [==============================] - 40s 802us/step - loss: 0.7312 - acc: 0.7416

Epoch 5/10

50000/50000 [==============================] - 40s 802us/step - loss: 0.4644 - acc: 0.8372

Epoch 6/10

50000/50000 [==============================] - 40s 801us/step - loss: 0.2792 - acc: 0.9023

Epoch 7/10

50000/50000 [==============================] - 40s 800us/step - loss: 0.1604 - acc: 0.9465

Epoch 8/10

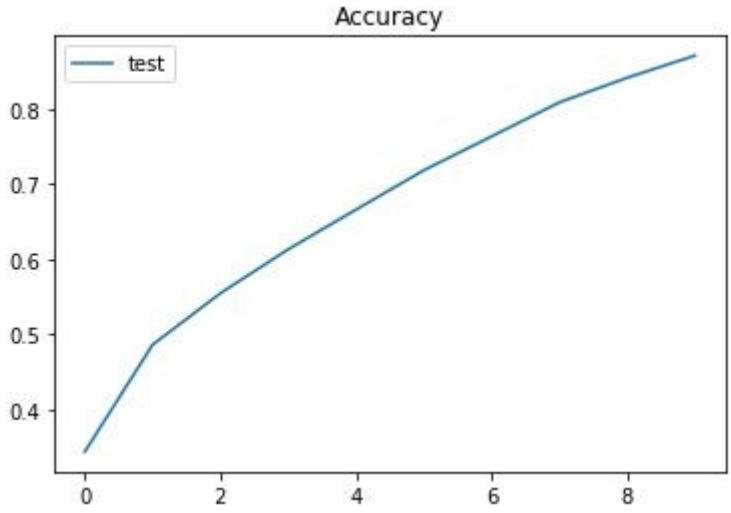
50000/50000 [==============================] - 40s 798us/step - loss: 0.1142 - acc: 0.9617

Epoch 9/10

50000/50000 [==============================] - 40s 798us/step - loss: 0.0979 - acc: 0.9676

Epoch 10/10

50000/50000 [==============================] - 40s 802us/step - loss: 0.0821 - acc: 0.9724

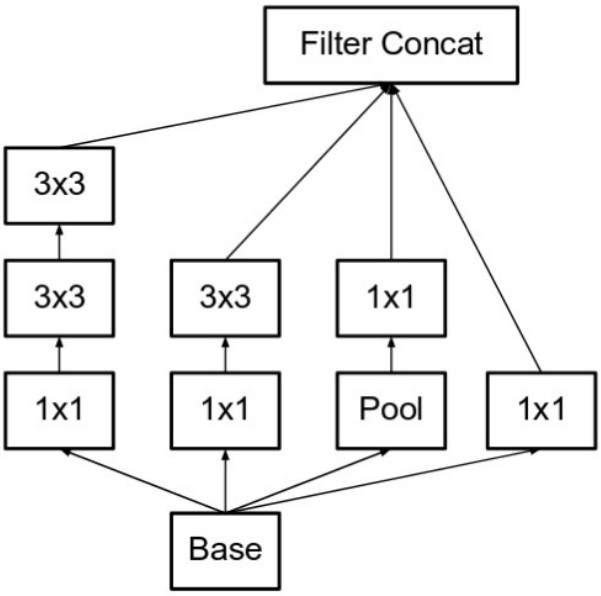


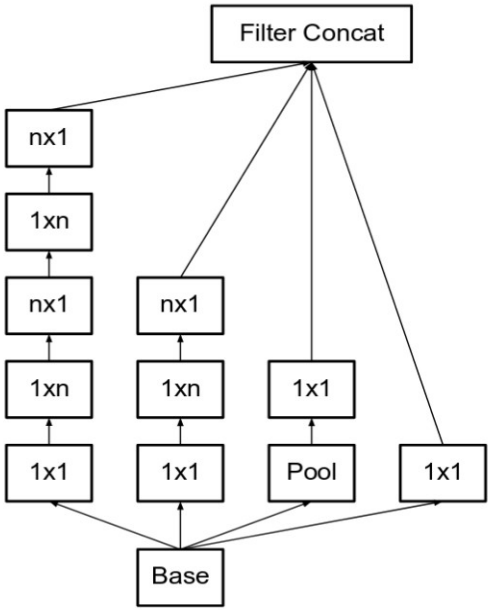
Test Accuracy:

0.62422763

C)Dimension Reduction (V2)

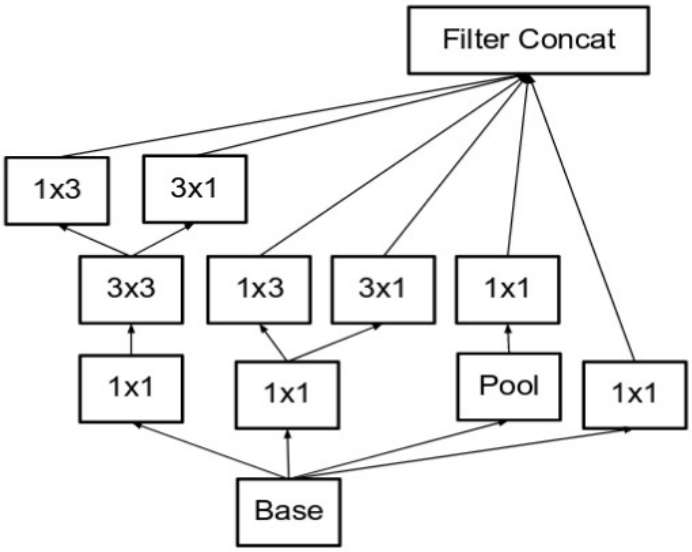
we implemented three architecture for this version shown below:





Architecture 1

Architecture 2



Architecture 3

Each of these architecture was built exactly like explained in part A & B with the same layer layout and concatenation of them.

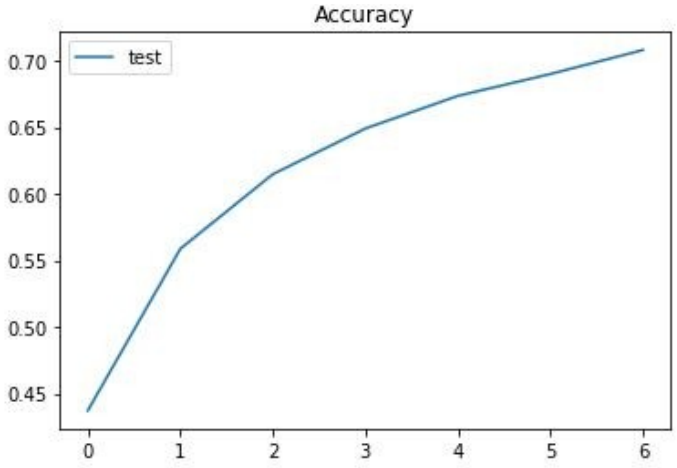
We then implemented this table found in the inception paper to represent the inception module Version 2

type	patch size/stride or remarks	input size
conv	$3\times3/2$	$299\times299\times3$
conv	$3\times3/1$	$149\times149\times32$
conv padded	$3\times3/1$	$147\times147\times32$
pool	$3\times3/2$	$147\times147\times64$
conv	$3\times3/1$	$73\times73\times64$
conv	$3\times3/2$	$71\times71\times80$
conv	$3\times3/1$	$35\times35\times192$
$3\times$Inception	As in figure 5	$35\times35\times288$
$5\times$Inception	As in figure 6	$17\times17\times768$
$2\times$Inception	As in figure 7	$8\times8\times1280$
pool	8×8	$8\times8\times2048$
linear	logits	$1\times1\times2048$
softmax	classifier	$1\times1\times1000$

The ‘3 Inception’ ‘5 x Inception’ & ‘2 x Inception’ rows
what was done was that the architecture 1 was repeated 3 times with each concatenated layer fed as input to the next layer of architecture 1.
Same goes with the following 2 rows.
Followed by an 8x8 pooling layer, flattening and softmax activation function.

The results for fitting this model were as follows:
7 epochs:

Train on 45000 samples, validate on 5000 samples
Epoch 1/7 45000/45000 [========] - 66s 1ms/step - loss: 1.5568 - acc: 0.4372
Epoch 2/7 45000/45000 [========] - 59s 1ms/step - loss: 1.2293 - acc: 0.5587
Epoch 3/7 45000/45000 [========] - 60s 1ms/step - loss: 1.0823 - acc: 0.6147
Epoch 4/7 45000/45000 [========] - 59s 1ms/step - loss: 0.9942 - acc: 0.6490
Epoch 5/7 45000/45000 [========] - 60s 1ms/step - loss: 0.9246 - acc: 0.6735
Epoch 6/7 45000/45000 [========] - 59s 1ms/step - loss: 0.8811 - acc: 0.6900
Epoch 7/7 45000/45000 [========] - 59s 1ms/step - loss: 0.8330 - acc: 0.7080



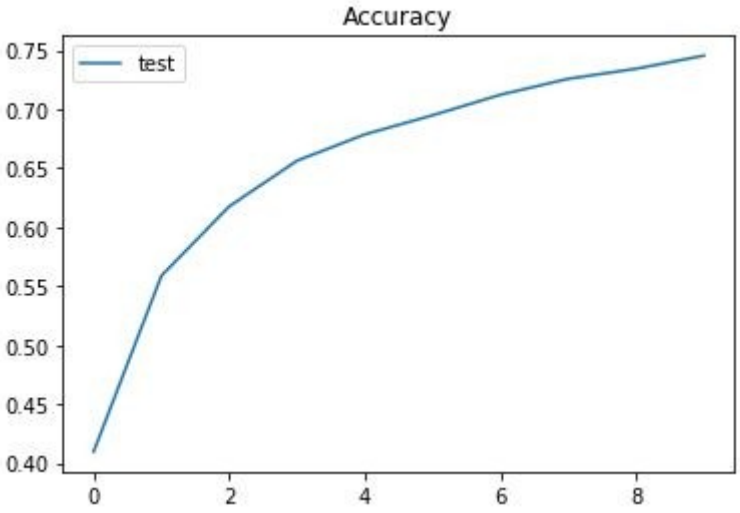
10000/10000 [==============================] - 4s 404us/step

Test Accuracy:

0.6799000030755997

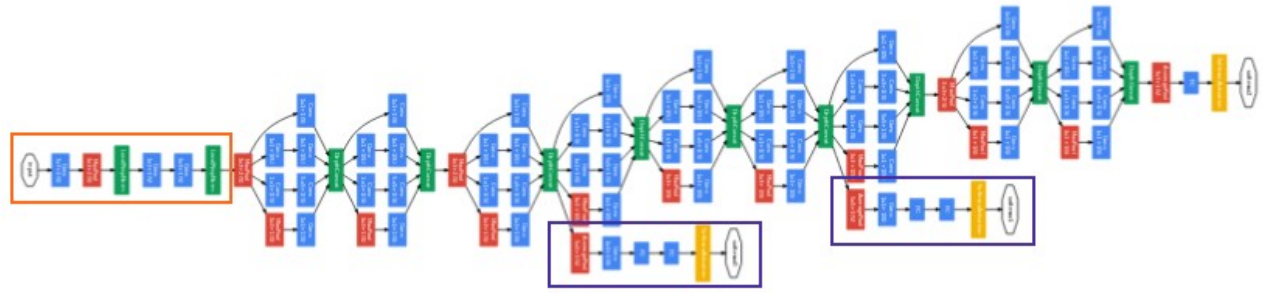
10 epochs:

Train on 45000 samples, validate on 5000 samples
Epoch 1/10 45000/45000 [==========] - 71s 2ms/step - loss: 1.6756 - acc: 0.4097
Epoch 2/10 45000/45000 [==========] - 60s 1ms/step - loss: 1.2446 - acc: 0.5589
Epoch 3/10 45000/45000 [==========] - 60s 1ms/step - loss: 1.0835 - acc: 0.6176
Epoch 4/10 45000/45000 [==========] - 60s 1ms/step - loss: 0.9795 - acc: 0.6564
Epoch 5/10 45000/45000 [==========] - 60s 1ms/step - loss: 0.9151 - acc: 0.6787
Epoch 6/10 45000/45000 [==========] - 60s 1ms/step - loss: 0.8704 - acc: 0.6949
Epoch 7/10 45000/45000 [==========] - 60s 1ms/step - loss: 0.8196 - acc: 0.7123
Epoch 8/10 45000/45000 [==========] - 60s 1ms/step - loss: 0.7800 - acc: 0.7258
Epoch 9/10 45000/45000 [==========] - 60s 1ms/step - loss: 0.7523 - acc: 0.7343
Epoch 10/10 45000/45000 [==========] - 60s 1ms/step - loss: 0.7233 - acc: 0.7454



test accuracy:
10000/10000 [==============================] - 4s 396us/step
0.6939000004529953

D) GoogleNet



For the googleNet implementation model we used the above picture to help us with the architecture.
GoogLeNet has 9 such inception modules stacked linearly. It is 22 layers deep (27, including the pooling layers). It uses global average pooling at the end of the last inception module.

The code follows the exact architecture shown above with dimension reduction functions used in between and 2 auxiliary classifiers.

The architecture ends with flattening and a fully connected neural network with 3 dense layers and a 0.5 dropout layer.

Fitting this model lead to the following results:

Epoch 1/10 50000/50000 [==============================] - 67s 1ms/step - loss: 2.8971 - dense_15_loss: 1.5238 - dense_30_loss: 1.5452 - dense_46_loss: 1.9764 - dense_15_acc: 0.4499 - dense_30_acc: 0.4432 - dense_46_acc: 0.3360
Epoch 2/10 50000/50000 [==============================] - 58s 1ms/step - loss: 2.2052 - dense_15_loss: 1.2407 - dense_30_loss: 1.2440 - dense_46_loss: 1.4598 - dense_15_acc: 0.5557 - dense_30_acc: 0.5567 - dense_46_acc: 0.4712
Epoch 3/10 50000/50000 [==============================] - 58s 1ms/step - loss: 1.9752 - dense_15_loss: 1.1187 - dense_30_loss: 1.1043 - dense_46_loss: 1.3083 - dense_15_acc: 0.6002 - dense_30_acc: 0.6041 - dense_46_acc: 0.5315
Epoch 4/10 50000/50000 [==============================] - 58s 1ms/step - loss: 1.8199 - dense_15_loss: 1.0312 - dense_30_loss: 1.0064 - dense_46_loss: 1.2086 - dense_15_acc: 0.6305 - dense_30_acc: 0.6418 - dense_46_acc: 0.5701
Epoch 5/10 50000/50000 [==============================] - 58s 1ms/step - loss: 1.6785 - dense_15_loss: 0.9516 - dense_30_loss: 0.9196 - dense_46_loss: 1.1172 - dense_15_acc: 0.6589 - dense_30_acc: 0.6708 - dense_46_acc: 0.6056
Epoch 6/10 50000/50000 [==============================] - 58s 1ms/step - loss: 1.5710 - dense_15_loss: 0.8765 - dense_30_loss: 0.8413 - dense_46_loss: 1.0557 - dense_15_acc: 0.6866 - dense_30_acc: 0.6989 - dense_46_acc: 0.6287
Epoch 7/10 50000/50000 [==============================] - 58s 1ms/step - loss: 1.4807 - dense_15_loss: 0.8110 - dense_30_loss: 0.7782 - dense_46_loss: 1.0039 - dense_15_acc: 0.7100 - dense_30_acc: 0.7212 - dense_46_acc: 0.6479

Epoch 8/10 50000/50000 [==============================] - 58s 1ms/step - loss: 1.4053 - dense_15_loss: 0.7497 - dense_30_loss: 0.7121 - dense_46_loss: 0.9668 - dense_15_acc: 0.7323 - dense_30_acc: 0.7451 - dense_46_acc: 0.6575
Epoch 9/10 50000/50000 [==============================] - 58s 1ms/step - loss: 1.3416 - dense_15_loss: 0.6929 - dense_30_loss: 0.6539 - dense_46_loss: 0.9376 - dense_15_acc: 0.7516 - dense_30_acc: 0.7649 - dense_46_acc: 0.6707
Epoch 10/10 50000/50000 [==============================] - 58s 1ms/step - loss: 1.2761 - dense_15_loss: 0.6438 - dense_30_loss: 0.5980 - dense_46_loss: 0.9036 - dense_15_acc: 0.7687 - dense_30_acc: 0.7858 - dense_46_acc: 0.6821
592.0250127315521 sec

10000/10000 [==============================] - 4s 360us/step

Test accuracy: 0.9315908338546753

Observations:

We concluded that the higher the version we implement the less computational power is needed.

However the time to fit the data would increase with higher versions naturally because more hidden layers were implemented in the network.

The accuracy also improves somewhat with higher versions and it depends mostly on the tuning of the parameters for each network
Parameters such as Batch size, Epochs, Number & size of filters, Dense layers & dropout. All played a significant role in the accuracy.

Most good results appeared from a 10 epochs run a batch size of 100 and a dropout of 0.5