Assignment 3 – CIFAR100 Classification Model

Dataset
We are using CIFAR100 from University of Toronto's website.
The CIFAR100 dataset consists of labeled subsets of 80million tiny images. This dataset comprises of 60000 32x32 color (or RGB) images. This dataset contains 100 classes (or categories) each comprising od 600 images.

The 100 classes in the CIFAR100 are grouped into 20 super-classes. Each image is associated with a "fine" label (the class it is associated with) and a "coarse" label (the superclass it is associated with). These super-classes and their corresponding sub-classes are tabulated below:

Superclass	Classes
aquatic mammals	beaver, dolphin, otter, seal, whale
fish	aquarium fish, flatfish, ray, shark, trout
flowers	orchids, poppies, roses, sunflowers, tulips
food containers	bottles, bowls, cans, cups, plates
fruit and vegetables	apples, mushrooms, oranges, pears, sweet peppers
household electrical devices	clock, computer keyboard, lamp, telephone, television
household furniture	bed, chair, couch, table, wardrobe
insects	bee, beetle, butterfly, caterpillar, cockroach
large carnivores	bear, leopard, lion, tiger, wolf
large man-made outdoor things	bridge, castle, house, road, skyscraper
large natural outdoor scenes	cloud, forest, mountain, plain, sea
large omnivores and herbivores	camel, cattle, chimpanzee, elephant, kangaroo
medium-sized mammals	fox, porcupine, possum, raccoon, skunk
non-insect invertebrates	crab, lobster, snail, spider, worm
people	baby, boy, girl, man, woman
reptiles	crocodile, dinosaur, lizard, snake, turtle
small mammals	hamster, mouse, rabbit, shrew, squirrel
trees	maple, oak, palm, pine, willow
vehicles 1	bicycle, bus, motorcycle, pickup truck, train
vehicles 2	lawn-mower, rocket, streetcar, tank, tractor

Problem Statement
In General, classification of objects in images is a difficult task due to factors such as, size of the object, lighting, occlusion, etc.
Classification using CIFAR100 is a challenging problem because there are many classes but the number of training samples for each class is very small. This makes it difficult obtain a high accuracy by training a model.

WINNING MODEL

Implementation
I have made use of Keras, a library built on top of TensorFlow to provide a higher-level API for facilitating and accelerating experiments for deep neural networks. Keras is fully compatible with TensorFlow as all its functions are built using tensors.

I also made use of Tensorboard, a web application for inspecting and understanding TensorFlow runs and graphs.

Load Data

I have used the keras import datasets. CIFAR100 to import the data. Next using the in-built function load_data, I load the testing and training data. Dataset consists of 50,000 32x32 color training images, labeled over 100 categories, and 10,000 test images.

Using the load_data() we get training data into a 4-dimensions tensor (number of records, image width, image length, number of channels). In addition, we put the labels of the testing data in the one-hot encoding format by using, keras.utils.to_categorical(y, num_classes=**None**);Converts a class vector (integers) to binary class matrix. Next, we split the X,Y into train and validation data. The split is set to 0.17. This means 17% of the data is used for validation, while the rest is used for training.

Architecture

My architecture consists of three convolutional layers, each layer uses a filter of size 3x3 and it uses a ReLU activation function to capture the non-linearity. Then, using the stride of step 1 (default for Keras Conv2D), and padding the input matrix with zeros around the border to apply filter to bordering elements of input image matrix.

Next, use max pooling layers along with our Conv2D of size 2x2 to arrive at scale invariant representation of the given image. Thus, this help us in detecting objects in the given image regardless of where it is located.

Lastly, using two fully connected layers, to get the final output. The first Fully connected layer (Dense in Keras) has size 512 nodes and using ReLU activiation function.

The last fully connected layer uses softmax activation function to get the probabilities of each class. The size = number of classes (100) nodes.To mitigate overfitting, we use a drop out of 50% in the last layer.

In the end using (loss = ‘categorical_crossentropy’) cross-entropy cost function and backpropagation to calculate the gradients of the error with respect to all weights in the network. Finally, I used Adam optimizer to update all filter values, weights and parameter values to minimize the output error.We can visualize the architecture by using keras.util.plot_model provided by keras.

CODE: Load Data

```
# Loda the cifar dataset
(X, Y), (x_test, y_test) = cifar100.load_data(label_mode='fine')


#split the X, Y in train and test samples
from sklearn.model_selection import train_test_split
x_train, x_val, y_train, y_val = train_test_split(X, Y,
                                                  test_size=0.17,
                                                  random_state=42)

print('x_train shape: ', x_train.shape)
print('train samples size: ', x_train.shape[0])
print( 'test samples size: ', x_test.shape[0])

# Convert class vectors to binary class matrices.
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
y_val = keras.utils.to_categorical(y_val, num_classes)
```

Results

CIFAR100 reached **56.02%** accuracy on the testing data while it reached 59% on training data.

Start of the Program

```
Using TensorFlow backend.
x_train shape:  (41500, 32, 32, 3)
train samples size:  41500
test samples size:  10000
Start Time is :  2017-11-28 14:47:18.254044
Using real-time data augmentation.
2017-11-28 14:47:18.498142: I tensorflow/core/platform/cpu_feature_guard.cc:137] Your CPU supports instru
as not compiled to use: SSE4.1 SSE4.2 AVX AVX2 FMA
2017-11-28 14:47:18.597790: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:892] successful NUMA
ue (-1), but there must be at least one NUMA node, so returning NUMA node zero
2017-11-28 14:47:18.598209: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1030] Found device 0 with
name: GeForce GTX 1080 major: 6 minor: 1 memoryClockRate(GHz): 1.7335
pciBusID: 0000:01:00.0
totalMemory: 7.92GiB freeMemory: 7.77GiB
2017-11-28 14:47:18.598235: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1120] Creating TensorFlow
, name: GeForce GTX 1080, pci bus id: 0000:01:00.0, compute capability: 6.1)
Epoch 1/150
1297/1297 [==============================] - 7s 6ms/step - loss: 4.3584 - acc: 0.0385 - val_loss: 4.0101
Epoch 2/150
1297/1297 [==============================] - 7s 5ms/step - loss: 3.9585 - acc: 0.0958 - val_loss: 3.6471
Epoch 3/150
1297/1297 [==============================] - 7s 5ms/step - loss: 3.7314 - acc: 0.1311 - val_loss: 3.4778
Epoch 4/150
1297/1297 [==============================] - 7s 5ms/step - loss: 3.5863 - acc: 0.1517 - val_loss: 3.3296
Epoch 5/150
1297/1297 [==============================] - 7s 5ms/step - loss: 3.4691 - acc: 0.1714 - val_loss: 3.2136
Epoch 6/150
```

End of the Program

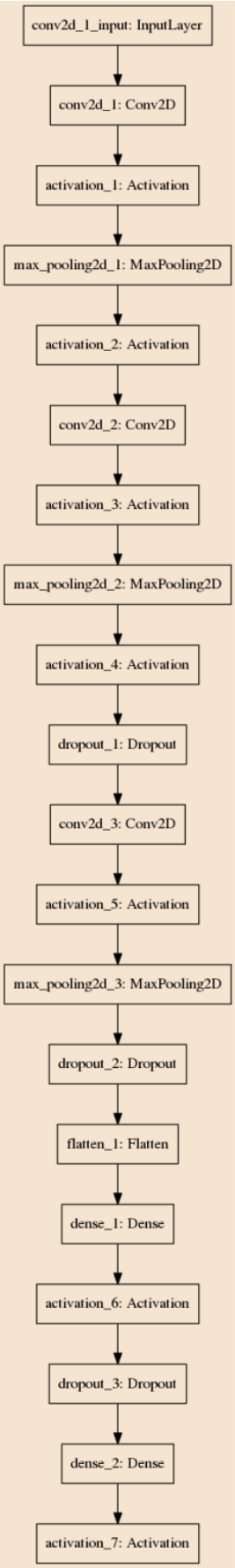
```
1297/1297 [==============================] - 7s 5ms/step - loss: 1.5819 - acc: 0.5561 - val_loss: 1.6796 - val_acc: 0.5474
Epoch 135/150
1297/1297 [==============================] - 7s 5ms/step - loss: 1.5834 - acc: 0.5529 - val_loss: 1.6825 - val_acc: 0.5462
Epoch 136/150
1297/1297 [==============================] - 7s 5ms/step - loss: 1.5727 - acc: 0.5572 - val_loss: 1.6785 - val_acc: 0.5447
Epoch 137/150
1297/1297 [==============================] - 7s 5ms/step - loss: 1.5769 - acc: 0.5573 - val_loss: 1.6692 - val_acc: 0.5486
Epoch 138/150
1297/1297 [==============================] - 7s 5ms/step - loss: 1.5586 - acc: 0.5615 - val_loss: 1.6655 - val_acc: 0.5488
Epoch 139/150
1297/1297 [==============================] - 7s 5ms/step - loss: 1.5676 - acc: 0.5593 - val_loss: 1.7014 - val_acc: 0.5425
Epoch 140/150
1297/1297 [==============================] - 7s 5ms/step - loss: 1.5589 - acc: 0.5616 - val_loss: 1.6743 - val_acc: 0.5480
Epoch 141/150
1297/1297 [==============================] - 7s 5ms/step - loss: 1.5576 - acc: 0.5594 - val_loss: 1.6557 - val_acc: 0.5520
Epoch 142/150
1297/1297 [==============================] - 7s 5ms/step - loss: 1.5504 - acc: 0.5627 - val_loss: 1.6783 - val_acc: 0.5467
Epoch 143/150
1297/1297 [==============================] - 7s 5ms/step - loss: 1.5449 - acc: 0.5648 - val_loss: 1.6720 - val_acc: 0.5467
Epoch 144/150
1297/1297 [==============================] - 7s 5ms/step - loss: 1.5423 - acc: 0.5642 - val_loss: 1.6610 - val_acc: 0.5474
Epoch 145/150
1297/1297 [==============================] - 7s 5ms/step - loss: 1.5415 - acc: 0.5624 - val_loss: 1.6708 - val_acc: 0.5479
Epoch 146/150
1297/1297 [==============================] - 7s 5ms/step - loss: 1.5377 - acc: 0.5649 - val_loss: 1.6521 - val_acc: 0.5496
Epoch 147/150
1297/1297 [==============================] - 7s 5ms/step - loss: 1.5305 - acc: 0.5657 - val_loss: 1.6522 - val_acc: 0.5522
```

```
End Time is :  2017-11-28 15:39:18.661231
Total time is : 0:16:41.926884
Saved trained model at /home/micha/divya/my_win/saved_models/model3_trained.h5
10000/10000 [==============================] - 1s 71us/step
Test loss: 1.65497318726
Test accuracy: 0.5602
```

CODE: Model

```
model.add(Conv2D(32, (3,3),
                 padding='same',
                 input_shape = input_dim))
model.add(Activation('relu'))

# MAXPOOL Layer 1
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Activation('relu'))

#CONV2D Layer 2
model.add(Conv2D(64, (3,3),
                 padding='same'))
model.add(Activation('relu'))

# MAXPOOL Layer 2
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Activation('relu'))

# Add a dropout of 10%
model.add(Dropout(0.1))

# CONV2D Layer 3
model.add(Conv2D(128,
         (3,3),
         padding='same'))
model.add(Activation('relu'))

# MAXPOOL Layer 3
model.add(MaxPooling2D(pool_size=(2,2)))

# Add dropout of 25%
model.add(Dropout(0.25))

# flatten
model.add(Flatten())

# Fully Connected Layer 1
model.add(Dense(512))
model.add(Activation('relu'))

# Adding a dropout of 50%
model.add(Dropout(0.5))

# Output Layer (Fully Connected Layer 2)
model.add(Dense(num_classes))
model.add(Activation('softmax'))
```

KERAS MODEL : Saved Model (model.PNG)

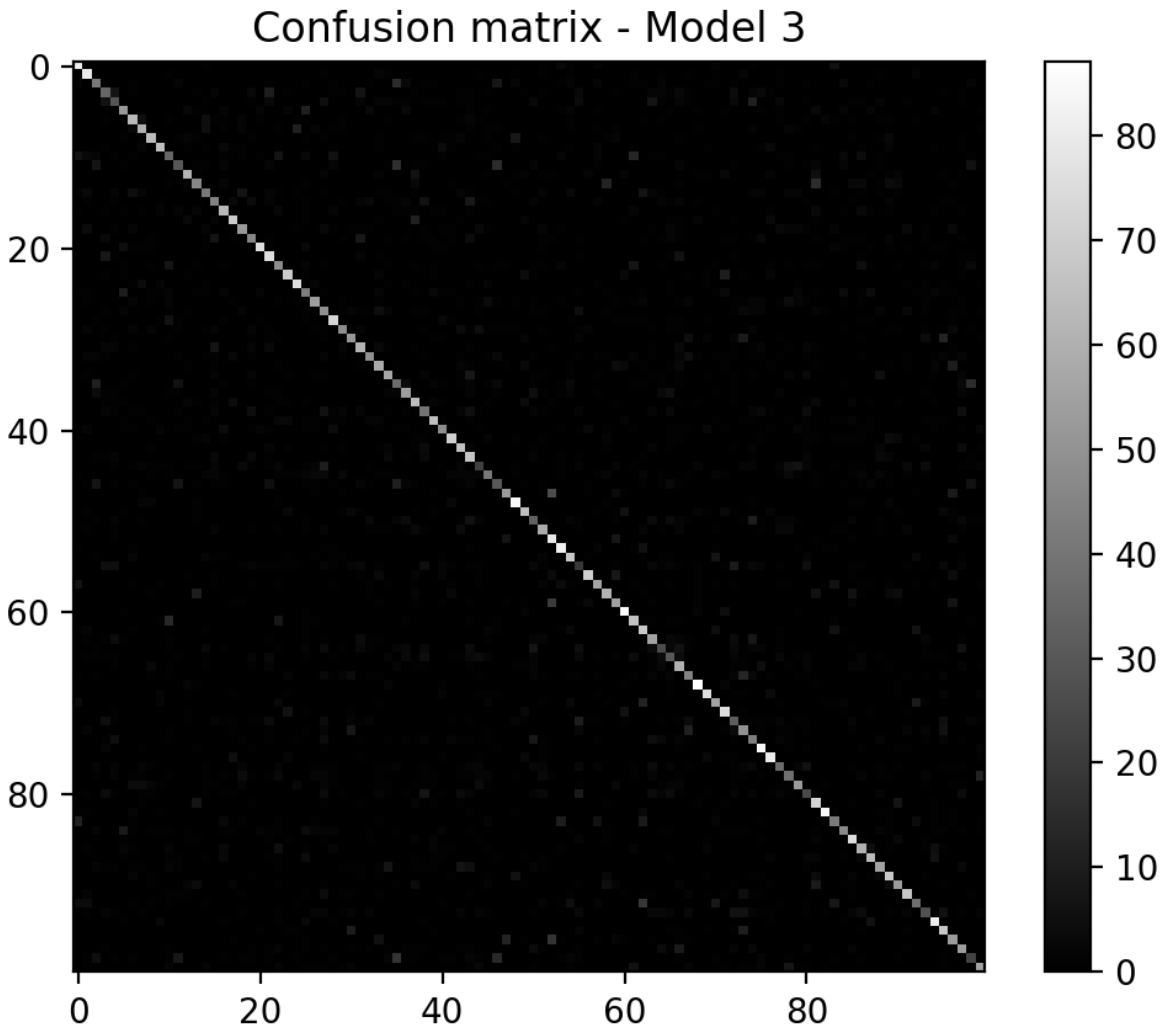


Confusion Matrix

A **confusion matrix** is a technique for summarizing the performance of a classification algorithm. The number of correct and incorrect predictions are summarized with count values and broken down by each class. This is the key to the confusion matrix.

The confusion matrix plotted below sows a clear diagonal: this shows that the true positives are high and false negatives are low, except for a few anomalies in the region around the diagonal where white spots appear. The confusion matrix supports the accuracy we have obtained.

In Grayscale:

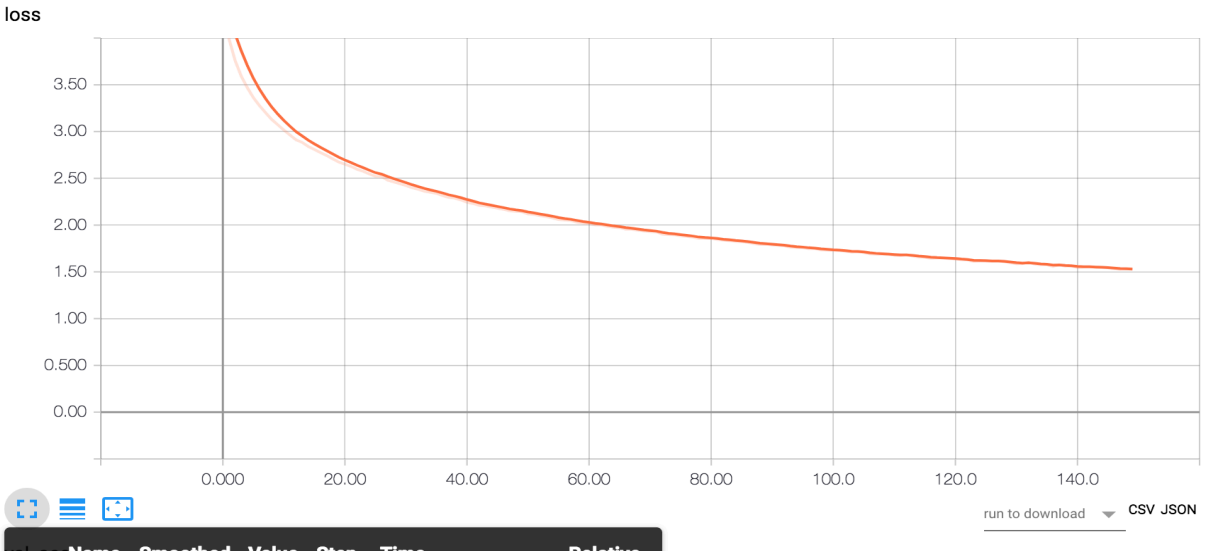


TensorBoard Graphs : Use tensorboard –logdir=Graph/ to visualize

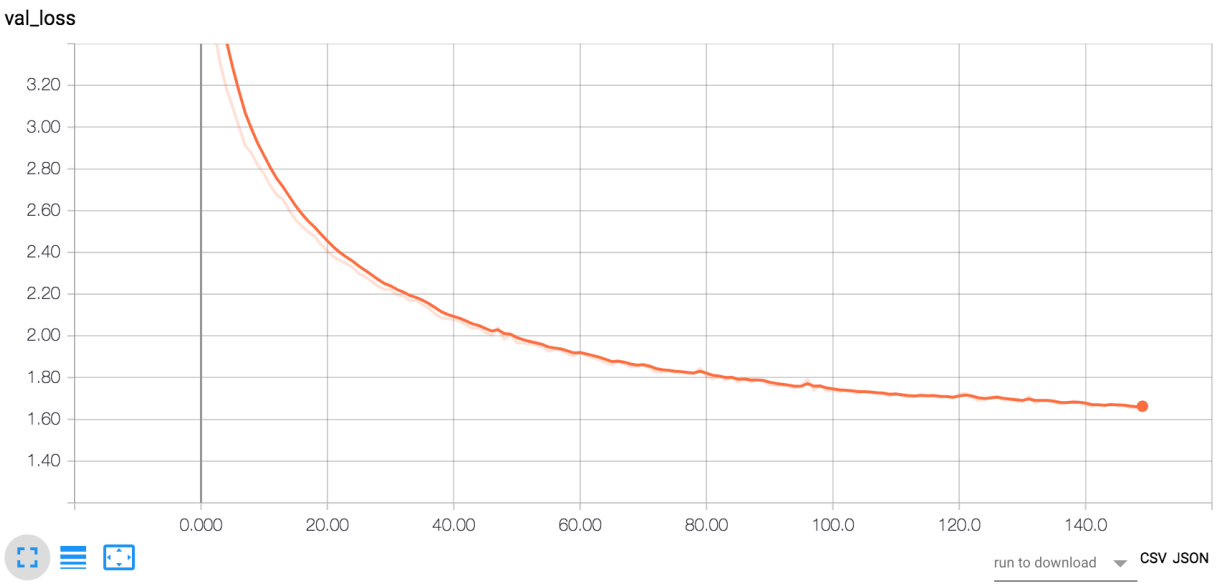
I made use of Tensorboard to obtain my loss, accuracy, val_loss and val_accuracy graphs. We need to add the following lines to enable the tensorboard while using keras,
tb_callback = keras.callbacks.TensorBoard(log_dir ='./Graph', histogram_freq = 0, write_graph = True, write_images=True)

In the keras.fit() set callbacks to list of callbacks to call during training.

Training Loss

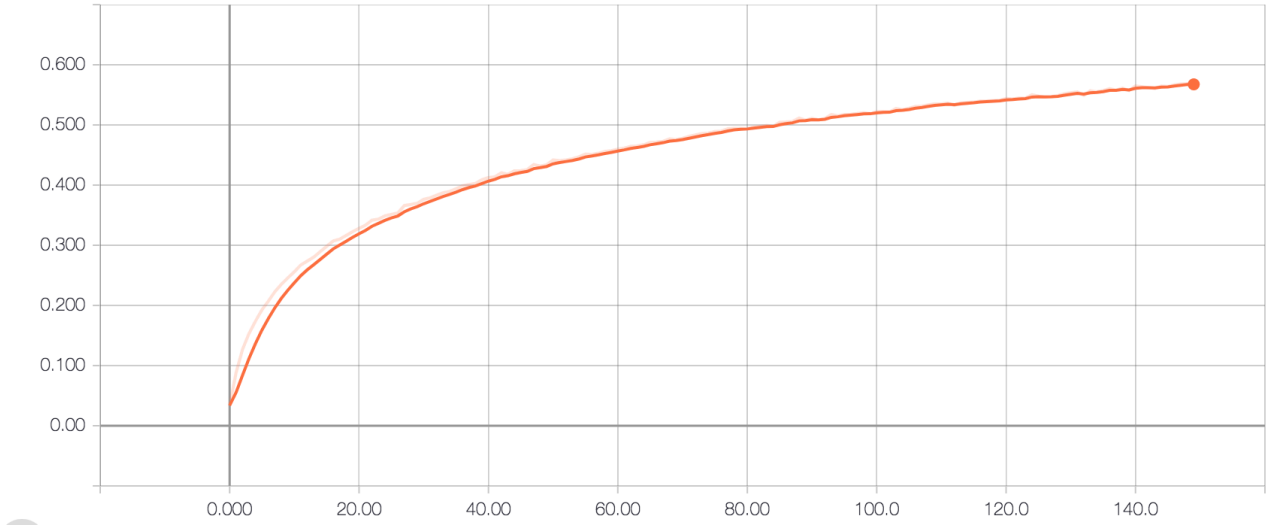


Validation Loss



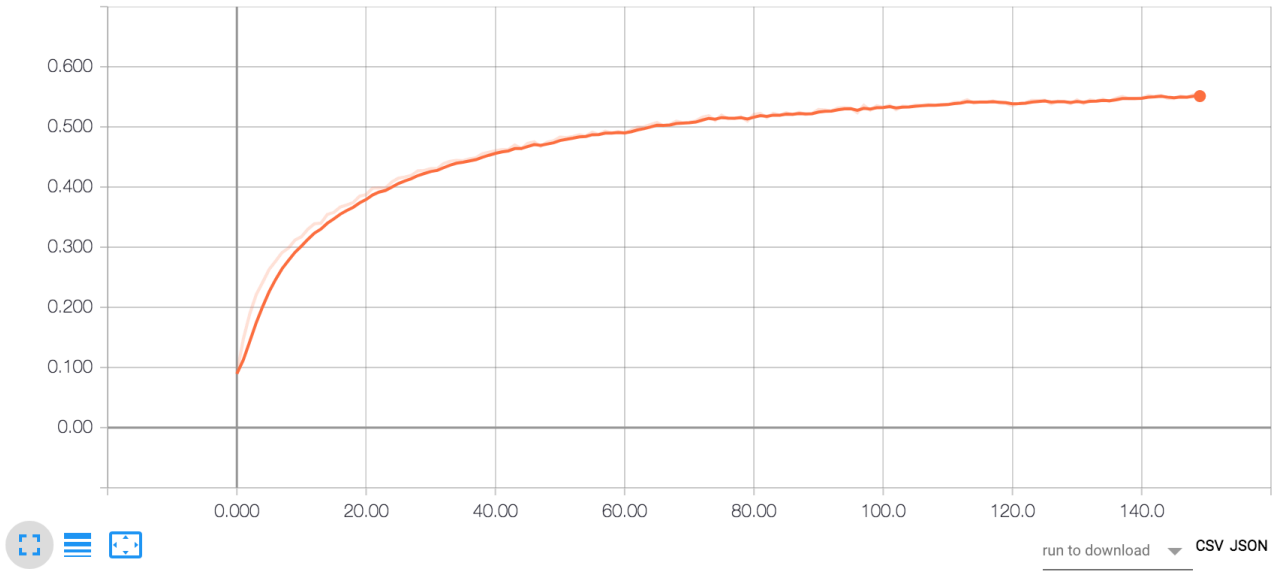
Training Accuracy

acc



Validation Accuracy

val_acc



Experiments

1. Initially, I started off with Adam optimizer set to learning rate of 0.001. The results showed no substantive convergence in the loss function. Then I reduced the initial learning rate to 0.0001. Now, this resulted in eventual convergence of the loss function.

2. I started off with the following architecture: shows in image model Architecture.

3. I decreased the dropout in the dropout layer 1 by 15% and increased it by 10% in dropout 3 (final dropout). This resulted in the model described above. This gave an increase of about 0.3% in test accuracy. This shows that slight change in the dropouts here and there can affect the model. We might be overfitting if the dropout is completely ignored.

Model with Dropout 1 = 0.25 and Dropout3 = 0.4

```
Epoch 80/100
50000/50000 [==============================] - 8s 169us/step - loss: 1.2432 - acc: 0.6379 - val_loss: 2.0350 - val_acc: 0.4853
Epoch 81/100
50000/50000 [==============================] - 12s 232us/step - loss: 1.2258 - acc: 0.6427 - val_loss: 2.0357 - val_acc: 0.4889
Epoch 82/100
50000/50000 [==============================] - 11s 227us/step - loss: 1.2153 - acc: 0.6425 - val_loss: 2.0305 - val_acc: 0.4875
Epoch 83/100
50000/50000 [==============================] - 11s 229us/step - loss: 1.2043 - acc: 0.6449 - val_loss: 2.0450 - val_acc: 0.4891
Epoch 84/100
50000/50000 [==============================] - 11s 221us/step - loss: 1.2018 - acc: 0.6480 - val_loss: 2.0446 - val_acc: 0.4862
Epoch 85/100
50000/50000 [==============================] - 11s 215us/step - loss: 1.1851 - acc: 0.6503 - val_loss: 2.0432 - val_acc: 0.4849
Epoch 86/100
50000/50000 [==============================] - 12s 234us/step - loss: 1.1743 - acc: 0.6520 - val_loss: 2.0450 - val_acc: 0.4897
Epoch 87/100
50000/50000 [==============================] - 11s 221us/step - loss: 1.1668 - acc: 0.6562 - val_loss: 2.0578 - val_acc: 0.4862
Epoch 88/100
50000/50000 [==============================] - 11s 229us/step - loss: 1.1565 - acc: 0.6554 - val_loss: 2.0687 - val_acc: 0.4847
Epoch 89/100
50000/50000 [==============================] - 11s 221us/step - loss: 1.1498 - acc: 0.6614 - val_loss: 2.0571 - val_acc: 0.4889
Epoch 90/100
50000/50000 [==============================] - 11s 226us/step - loss: 1.1406 - acc: 0.6614 - val_loss: 2.0592 - val_acc: 0.4903
Epoch 91/100
50000/50000 [==============================] - 11s 215us/step - loss: 1.1284 - acc: 0.6655 - val_loss: 2.0654 - val_acc: 0.4866
Epoch 92/100
50000/50000 [==============================] - 12s 230us/step - loss: 1.1221 - acc: 0.6653 - val_loss: 2.0776 - val_acc: 0.4856
Epoch 93/100
50000/50000 [==============================] - 10s 208us/step - loss: 1.0975 - acc: 0.6715 - val_loss: 2.0702 - val_acc: 0.4894
Epoch 94/100
50000/50000 [==============================] - 11s 228us/step - loss: 1.0997 - acc: 0.6708 - val_loss: 2.0623 - val_acc: 0.4901
Epoch 95/100
50000/50000 [==============================] - 12s 243us/step - loss: 1.0917 - acc: 0.6706 - val_loss: 2.0744 - val_acc: 0.4912
Epoch 96/100
50000/50000 [==============================] - 11s 223us/step - loss: 1.0730 - acc: 0.6769 - val_loss: 2.0735 - val_acc: 0.4900
Epoch 97/100
50000/50000 [==============================] - 11s 224us/step - loss: 1.0713 - acc: 0.6804 - val_loss: 2.0689 - val_acc: 0.4876
Epoch 98/100
50000/50000 [==============================] - 11s 210us/step - loss: 1.0599 - acc: 0.6824 - val_loss: 2.1000 - val_acc: 0.4898
Epoch 99/100
50000/50000 [==============================] - 12s 238us/step - loss: 1.0645 - acc: 0.6804 - val_loss: 2.0830 - val_acc: 0.4900
Epoch 100/100
50000/50000 [==============================] - 11s 218us/step - loss: 1.0459 - acc: 0.6829 - val_loss: 2.0849 - val_acc: 0.4917
End Time is :  2017-11-26 18:02:36.159232
Total time is : 0:17:56.435129
Saved trained model at /home/micha/divya/saved_models/model2_test1_trained.h5
10000/10000 [==============================] - 1s 80us/step
Test loss: 2.08494180679
Test accuracy: 0.4917
```

4. I also tried experimenting with the decay value of the ADAM optimizer. By setting decay = 0.05. The test accuracy dropped steeply to 11.9%, which is huge.

*lr = self.lr * (1. / (1. + self.decay * self.iterations))*

My reasoning to justify this steep decrease is that increasing the decay causes the learning rate to change very steeply thus no more gradient learning. This causes the accuracy to drop and losses never decrease.

```
50000/50000 [==============================] - 11s 228us/step - loss: 4.0371 - acc: 0.0855 - val_loss: 3.9354 - val_acc: 0.1192
Epoch 95/100
50000/50000 [==============================] - 11s 220us/step - loss: 4.0352 - acc: 0.0861 - val_loss: 3.9346 - val_acc: 0.1188
Epoch 96/100
50000/50000 [==============================] - 12s 244us/step - loss: 4.0348 - acc: 0.0882 - val_loss: 3.9337 - val_acc: 0.1191
Epoch 97/100
50000/50000 [==============================] - 11s 218us/step - loss: 4.0318 - acc: 0.0866 - val_loss: 3.9327 - val_acc: 0.1197
Epoch 98/100
50000/50000 [==============================] - 12s 230us/step - loss: 4.0324 - acc: 0.0863 - val_loss: 3.9318 - val_acc: 0.1196
Epoch 99/100
50000/50000 [==============================] - 11s 218us/step - loss: 4.0341 - acc: 0.0850 - val_loss: 3.9309 - val_acc: 0.1200
Epoch 100/100
50000/50000 [==============================] - 12s 232us/step - loss: 4.0288 - acc: 0.0852 - val_loss: 3.9301 - val_acc: 0.1195
End Time is :  2017-11-26 18:48:21.457353
Total time is : 0:18:19.527006
Saved trained model at /home/micha/divya/saved_models/model2_test2_trained.h5
10000/10000 [==============================] - 1s 74us/step
Test loss: 3.93013969345
Test accuracy: 0.1195
```

5. SGD Error Model. After reading about SGD and Adam online, I understand that,

Adam : ".. many objective functions are composed of a sum of sub-functions evaluated at different subsamples of data; int this case optimization can be made more efficient by taking gradient steps wrt individual sub-functions..." What this means is that, the objective of the function is the sum of errors over training samples. The training can be done on individual samples or in small batches.

SGD : Stochastic Gradient descent is very similar to the working of Adam. The major difference being SGD is more efficient for large scale problems than batch training because parameter updates are more frequent. Thus, in case of CIFAR100, where the number of images in each sub-class are very small, Adam outperforms SGD. This is the reason for SGD showing no learning rate or minimum accuracy here.

```
41500/41500 [==============================] - 9s 225us/step - loss: 4.5968 - acc: 0.0141 - val_loss: 4.5957 - val_acc: 0.0227
Epoch 92/100
41500/41500 [==============================] - 11s 271us/step - loss: 4.5962 - acc: 0.0143 - val_loss: 4.5955 - val_acc: 0.0228
Epoch 93/100
41500/41500 [==============================] - 11s 263us/step - loss: 4.5960 - acc: 0.0147 - val_loss: 4.5953 - val_acc: 0.0225
Epoch 94/100
41500/41500 [==============================] - 10s 250us/step - loss: 4.5959 - acc: 0.0144 - val_loss: 4.5952 - val_acc: 0.0228
Epoch 95/100
41500/41500 [==============================] - 10s 252us/step - loss: 4.5958 - acc: 0.0145 - val_loss: 4.5950 - val_acc: 0.0224
Epoch 96/100
41500/41500 [==============================] - 11s 264us/step - loss: 4.5955 - acc: 0.0141 - val_loss: 4.5948 - val_acc: 0.0226
Epoch 97/100
41500/41500 [==============================] - 9s 227us/step - loss: 4.5952 - acc: 0.0152 - val_loss: 4.5946 - val_acc: 0.0231
Epoch 98/100
41500/41500 [==============================] - 10s 247us/step - loss: 4.5953 - acc: 0.0140 - val_loss: 4.5944 - val_acc: 0.0231
Epoch 99/100
41500/41500 [==============================] - 11s 273us/step - loss: 4.5948 - acc: 0.0157 - val_loss: 4.5942 - val_acc: 0.0235
Epoch 100/100
41500/41500 [==============================] - 11s 260us/step - loss: 4.5953 - acc: 0.0149 - val_loss: 4.5940 - val_acc: 0.0238
End Time is :  2017-11-26 22:55:59.245313
Total time is : 0:16:21.258586
Saved trained model at /home/micha/divya/saved_models_test3/model1_test3_trained.h5
10000/10000 [==============================] - 1s 108us/step
Test loss: 4.59286362
Test accuracy: 0.0211
```

6. Finally, all the models described above did not use data augmentation. I started off with building a simple CNN with three conv2D layers, three dropout layers of 0.25, 0.25 and .050 respectively. Then, I went on to add 3 maxPool layers and then added the last two fully connected layers. This model resulted in the second best accuracy on the Test-Data: **49.50%**
Upon adding data augmentation to this model, I got my winning model. Just adding Image preprocessing code to my model gave a boost of about **5%** (final is 56.05). To my understanding, there was an increase in the accuracy due to more number of images to train on after doing data augmentation.

```
50000/50000 [==============================] - 14s 285us/step - loss: 1.2494 - acc: 0.6347 - val_loss: 2.0193 - val_acc: 0.4916
Epoch 87/100
50000/50000 [==============================] - 14s 283us/step - loss: 1.2323 - acc: 0.6360 - val_loss: 2.0331 - val_acc: 0.4902
Epoch 88/100
50000/50000 [==============================] - 13s 254us/step - loss: 1.2335 - acc: 0.6363 - val_loss: 2.0380 - val_acc: 0.4891
Epoch 89/100
50000/50000 [==============================] - 14s 280us/step - loss: 1.2176 - acc: 0.6428 - val_loss: 2.0340 - val_acc: 0.4936
Epoch 90/100
50000/50000 [==============================] - 13s 263us/step - loss: 1.2081 - acc: 0.6432 - val_loss: 2.0233 - val_acc: 0.4908
Epoch 91/100
50000/50000 [==============================] - 14s 281us/step - loss: 1.1976 - acc: 0.6470 - val_loss: 2.0329 - val_acc: 0.4890
Epoch 92/100
50000/50000 [==============================] - 13s 263us/step - loss: 1.1946 - acc: 0.6458 - val_loss: 2.0376 - val_acc: 0.4900
Epoch 93/100
50000/50000 [==============================] - 13s 266us/step - loss: 1.1820 - acc: 0.6517 - val_loss: 2.0250 - val_acc: 0.4937
Epoch 94/100
50000/50000 [==============================] - 14s 273us/step - loss: 1.1655 - acc: 0.6574 - val_loss: 2.0330 - val_acc: 0.4905
Epoch 95/100
50000/50000 [==============================] - 13s 264us/step - loss: 1.1570 - acc: 0.6577 - val_loss: 2.0242 - val_acc: 0.4927
Epoch 96/100
50000/50000 [==============================] - 14s 273us/step - loss: 1.1461 - acc: 0.6604 - val_loss: 2.0360 - val_acc: 0.4915
Epoch 97/100
50000/50000 [==============================] - 14s 279us/step - loss: 1.1450 - acc: 0.6580 - val_loss: 2.0432 - val_acc: 0.4874
Epoch 98/100
50000/50000 [==============================] - 13s 251us/step - loss: 1.1283 - acc: 0.6634 - val_loss: 2.0434 - val_acc: 0.4918
Epoch 99/100
50000/50000 [==============================] - 14s 284us/step - loss: 1.1361 - acc: 0.6642 - val_loss: 2.0376 - val_acc: 0.4932
Epoch 100/100
50000/50000 [==============================] - 14s 274us/step - loss: 1.1241 - acc: 0.6652 - val_loss: 2.0539 - val_acc: 0.4942
End Time is :  2017-11-26 14:01:45.101932
Total time is : 0:21:57.039830
Saved trained model at /home/micha/divya/saved_models/keras_cifar10_trained_model.h5
10000/10000 [==============================] - 1s 132us/step
Test loss: 2.05392160969
Test accuracy: 0.4942
micha@neu:~/divya$
```