Deep learning- assignment 1

Dataset selected CIFAR-10

1. a. the data contain 60000 32x32 color images, 6000 images per class. There are 50000 training images and 10000 test images.

   b. each sample represent a 32x32 color image, so each sample built from 3072 values (pixels). The first 1024 entries contain the red channel values, the next 1024 the green, and the final 1024 the blue (each image have three channels).
   At first look, we thought that we do not need to preprocess the data- it is ready to use. After the first runs and reading about improvement of the model, we normalized the data to increase the accuracy of the model. Therefore, the first run is without normalization.
   We can use augmentation but for the beginning, we want to see the accuracy on the test set without changing the original data. When we going to do augmentation we can do horizontal_flip but we cannot do vertical_flip because it is not appropriate for most of the category: automobile, cats, dear, dog, horse, ship and track. For example for ships, vertical_flip produce an image that do not represent realistic situation. Therefore, we do not want the model to learn from this example:

   We can use a small range of rotation (e.g. 10-15 degree) because more than this will not reflect realistic situation for the category mention above.
   Summary:

dimensions	32x32
channels	3 (rgb)
Number of class	10
values	0-255

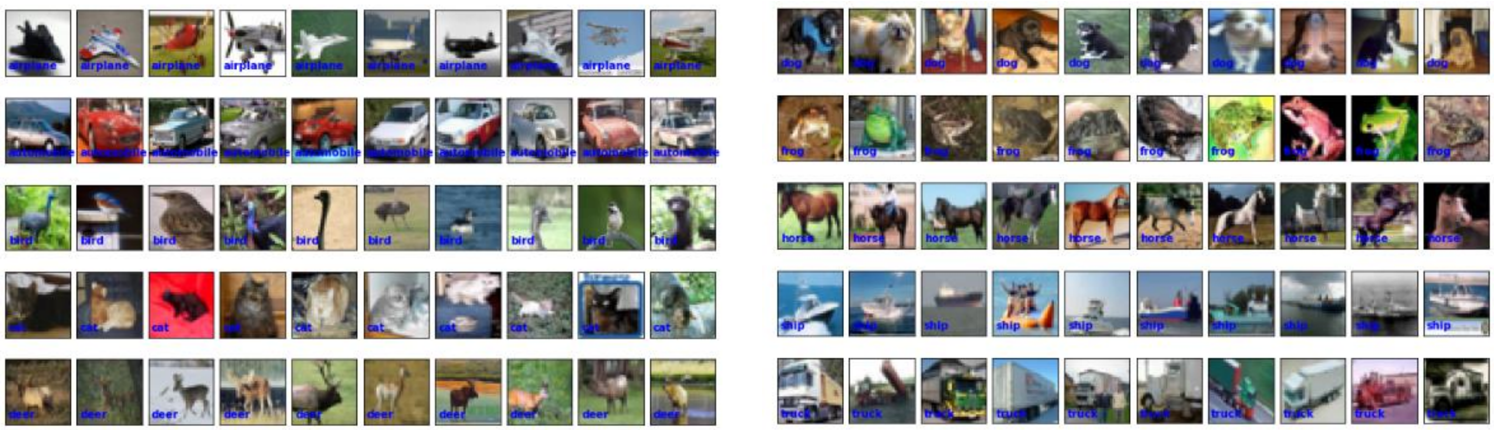
   c. the data is balanced- there is exactly 5000 samples in each class:

```
Label Counts of [0](AIRPLANE) : 5000
Label Counts of [1](AUTOMOBILE) : 5000
Label Counts of [2](BIRD) : 5000
Label Counts of [3](CAT) : 5000
Label Counts of [4](DEER) : 5000
Label Counts of [5](DOG) : 5000
Label Counts of [6](FROG) : 5000
Label Counts of [7](HORSE) : 5000
Label Counts of [8](SHIP) : 5000
Label Counts of [9](TRUCK) : 5000
```

   d. yes, on a brief search online we find a lot of works on the CIFAR-10 dataset.

The best result we find (best accuracy) is with 96.53% accuracy on test set. It uses 100 passes at test time. Reaches 95.5% when using a single pass at test time, and 96.33% when using 12 passes. Uses data augmentation during training.

e. samples from each label:



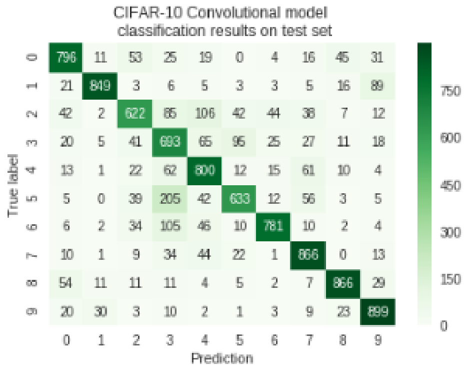
2. a. validation strategy: we use Train-test split by the train and test set we get from https://competitions.codalab.org/competitions/19854 so our train set contain all the 5 batches and the test set contain the test set provided. Therefore, our train set contain 50000 samples and our test set contain 10000 samples.

   b. first results: ~78% test accuracy and ~84% train accuracy, for 10 epochs, with the next model:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 30, 30, 32)        896
_________________________________________________________________
batch_normalization_1 (Batch (None, 30, 30, 32)        128
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 28, 28, 32)        9248
_________________________________________________________________
batch_normalization_2 (Batch (None, 28, 28, 32)        128
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 32)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 12, 12, 64)        18496
_________________________________________________________________
batch_normalization_3 (Batch (None, 12, 12, 64)        256
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 10, 10, 64)        36928
_________________________________________________________________
batch_normalization_4 (Batch (None, 10, 10, 64)        256
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 5, 5, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1600)              0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                16010
=================================================================
Total params: 82,346
Trainable params: 81,962
Non-trainable params: 384
```

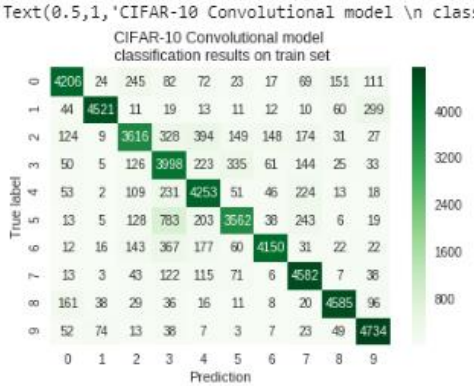
test accuracy:

```
model accuracy on test set is: 78.05%
Text(0.5,1,'CIFAR-10 Convolutional model \n classification results on test set')
```



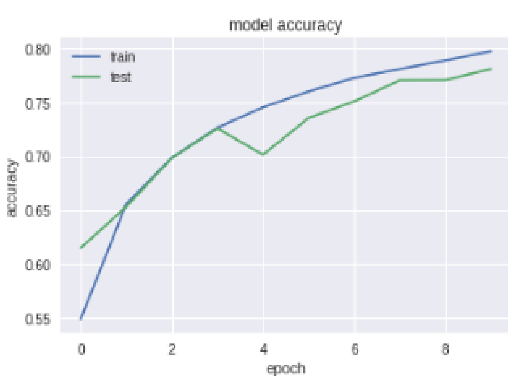
train accuracy

```
model accuracy on train set is: 84.414%
Text(0.5,1,'CIFAR-10 Convolutional model \n classification results on train set')
```



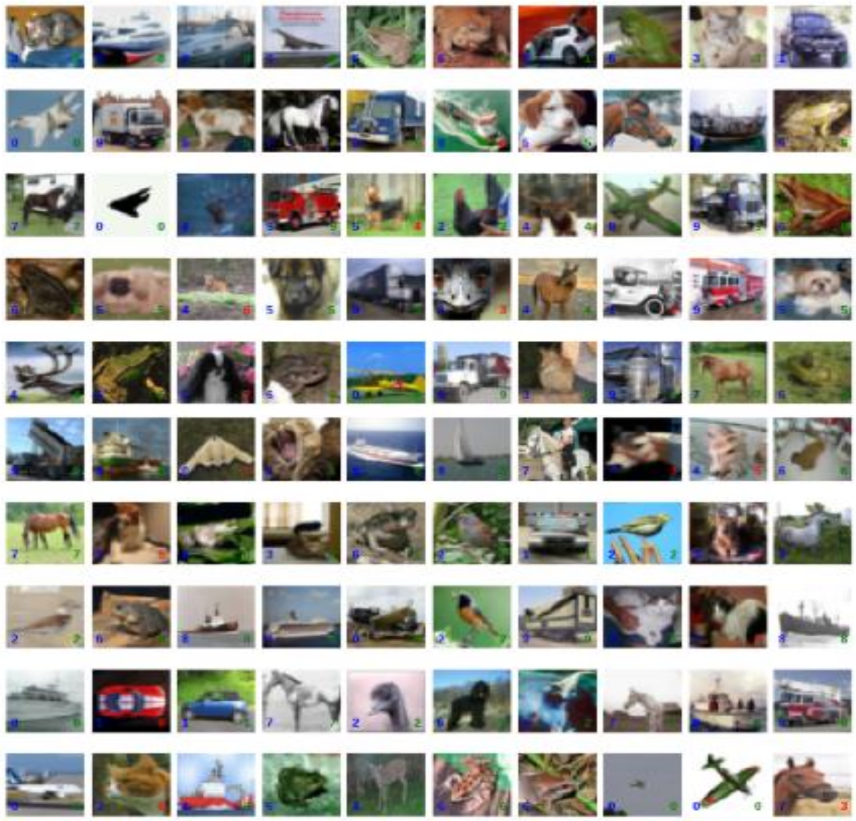
test-train accuracy:

Note: train set accuracy is ~6% higher than test accuracy, therefore we suspect overfitting



Example of good and bad classification:

Note: the class is sign with number, when blue number is the labeled class, green number is the right prediction and red number is wrong prediction.



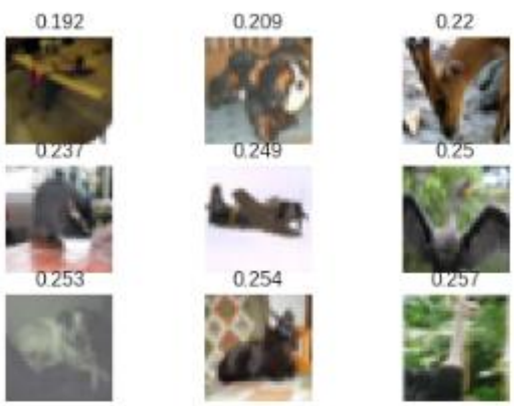
examples of good classification with lowest probability:

pred_prob	pred_cat	y_test_cat	idx	class
0.210037	3	3	5835	good
0.235577	2	2	8454	good
0.244407	6	6	1580	good
0.262622	3	3	8966	good
0.268713	5	5	1042	good
0.272566	5	5	6116	good
0.280755	6	6	6656	good
0.281085	5	5	7787	good
0.281658	7	7	5855	good

examples of good classification with highest probability:

pred_prob	pred_cat	y_test_cat	idx	class
1.0	9	9	7844	good
1.0	6	6	2445	good
1.0	1	1	6048	good
1.0	1	1	2947	good
1.0	8	8	9655	good
1.0	1	1	8157	good
1.0	8	8	2713	good
1.0	1	1	8839	good
1.0	1	1	8522	good

examples of bad classification with lowest probability:



pred_prob	pred_cat	y_test_cat	idx	class
0.192046	7	0	8476	bad
0.208636	9	5	2273	bad
0.219951	9	4	6210	bad
0.237365	7	3	4175	bad
0.249146	6	0	2463	bad
0.250125	3	2	5129	bad
0.253148	4	5	8236	bad
0.253667	4	3	3602	bad
0.256531	3	2	3474	bad

examples of bad classification with highest probability:

pred_prob	pred_cat	y_test_cat	idx	class
0.998883	6	3	6213	bad
0.998925	1	9	3501	bad
0.998942	1	9	9817	bad
0.998956	9	1	7311	bad
0.998990	8	9	2495	bad
0.999013	8	0	6588	bad
0.999256	6	3	5511	bad
0.999388	9	0	7861	bad
0.999963	6	3	2405	bad

c. we think that the main resons for misclassifying:

i. First, the most unusual phenomenon we have identified is that the most common mistake is that the model is mistakenly classified images from all kind of class as frog. We think is because picture of a frog usually contains an unclear shape that takes up a large part of the size of the picture. The background can be water (similar to the sky), grass, road, etc. Therefore, to make a clearer separation between the objects we thought we might normalize the data in order to make a clearer distinction between textures, shapes, etc.

ii. as we can see, 'automobile' image classification is wrong 98 times and classified as 'truck'- we think we need to use augmentation so the model would learn from more examples of automobile (we suppose that same images in different positions will have the same impact as more automobile images).

iii. as we discuses in (ii) we can also see that "cat" image classification is wrong 145 times and classified as "dog". we suggest augmentation in this case as well

Ways to improve the results:

1. augmentation
2. normalization of the data
3. need more layers\ other parameters.
4. change optimizer
5. add momentum
6. add epochs

d. Prioritize the list from 2.c:

1. **normalization of the data**
2. **augmentation**
3. add epochs
4. add momentum
5. change optimizer
6. need more layers\ other parameters.

We think that we going to get the best improvement with applying 1 and 2 suggestions from the list above.

As we can see, from the train-test accuracy histogram, the graph of the train and the test is also increasing in the tenth epoch, which means that it has not yet reached a situation where the model accuracy is not improved. Therefore, in our opinion, increasing the number of epochs may improve the accuracy of the model.

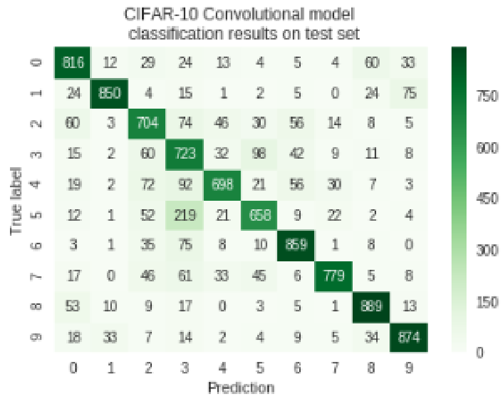
- After only normalization of the data using z-score normalization, we get improvement of ~ 0.5% on test set accuracy and ~1% on train set accuracy: ~78% test accuracy and ~85% train accuracy:

  Normalization:

```
x_train = x_train.astype("float32")
x_test = x_test.astype("float32")
mean = np.mean(x_train)
std = np.std(x_train)
x_train = (x_train - mean) / std
x_test = (x_test - mean) / std
```

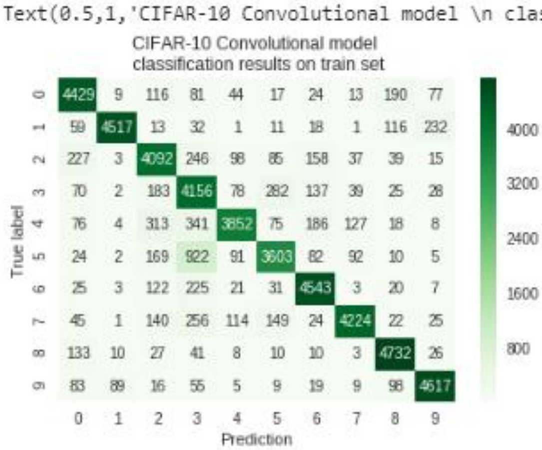
test accuracy:

```
model accuracy on test set is: 78.5%
Text(0.5,1,'CIFAR-10 Convolutional model \n classification results on test set')
```



train accuracy:

```
model accuracy on train set is: 85.53%
Text(0.5,1,'CIFAR-10 Convolutional model \n classification results on train set')
```



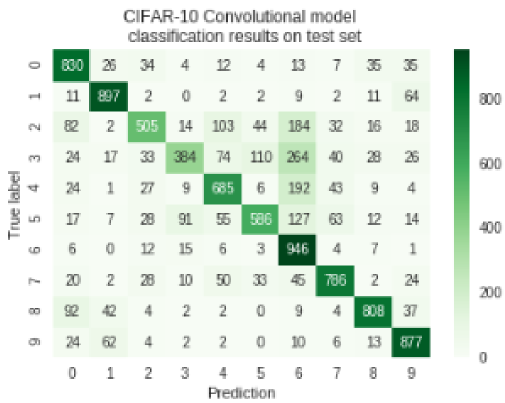
train-test accuracy:

- After normalization and augmentation we get Deterioration test and train accuracy, ~73% test accuracy and ~75% train accuracy:
  augmentation parametars:

```
#data augmentation
datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True,
    )
datagen.fit(x_train)
```

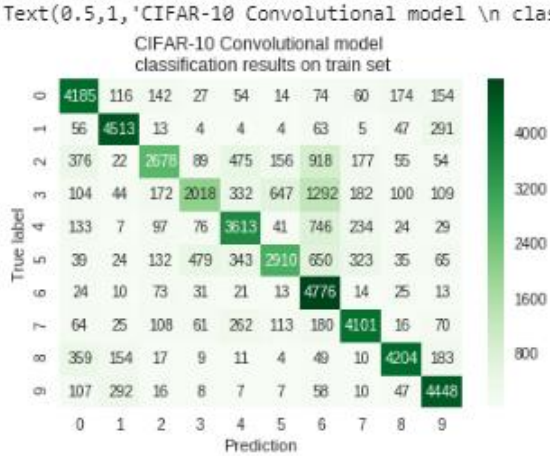
test accuracy:

```
model accuracy on test set is: 73.04%
Text(0.5,1,'CIFAR-10 Convolutional model \n classification results on test set')
```



train accuracy:

```
model accuracy on train set is: 74.892%
Text(0.5,1,'CIFAR-10 Convolutional model \n classification results on train set')
```



- After reading "MACHINE LEARNING IN ACTION" (mainly part 3) we decide we need to add Regularization technique to improve the result of our model, to speed up the training process and prevent over-fitting. Therefore, we add changes in learning rate as callback and kernel regularizer. We also increased the number of epochs from 10 to 100:

- for the learning rate, we add two points to change the learning rate:

```
def lr_schedule(epoch):
    lrate = 0.001
    if epoch > 50:
        lrate = 0.0005
    elif epoch > 75:
        lrate = 0.0003
    return lrate
```

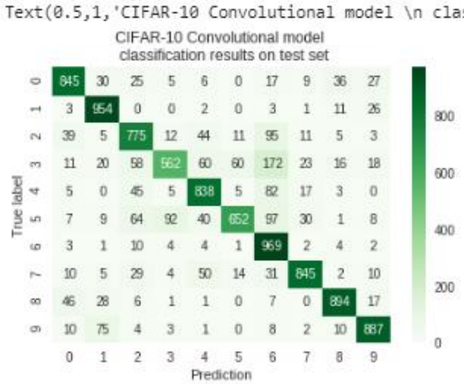
- for kernel regularizer we add l2 regularizer with 1e-4 weight decay (for each convolution layer):

```
weight_decay = 1e-4
model = Sequential()
model.add(Conv2D(32, (3,3), activation='relu', kernel_regularizer=regularizers.l2(weight_decay), input_shape=x_train.shape[1:]))
```

We get improvement in test and train accuracy: 82% test accuracy and ~86% train accuracy:

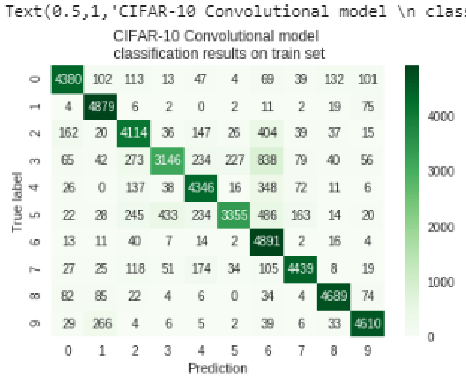
<u>test accuracy:</u>

```
model accuracy on test set is: 82.21000000000001%
Text(0.5,1,'CIFAR-10 Convolutional model \n classification results on test set')
```

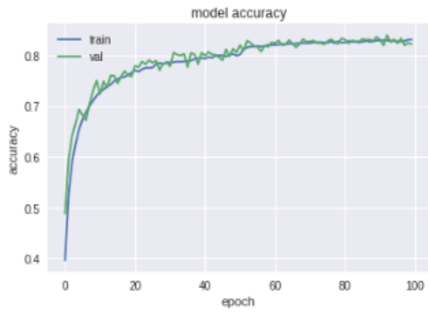


<u>train accuracy:</u>

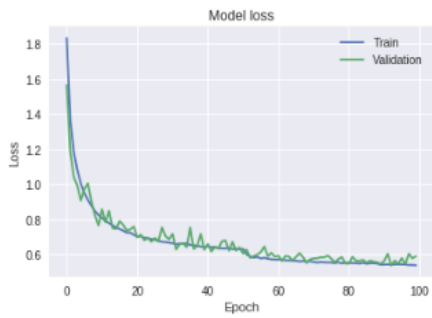
```
model accuracy on train set is: 85.698%
Text(0.5,1,'CIFAR-10 Convolutional model \n classification results on train set')
```



train-test accuracy:



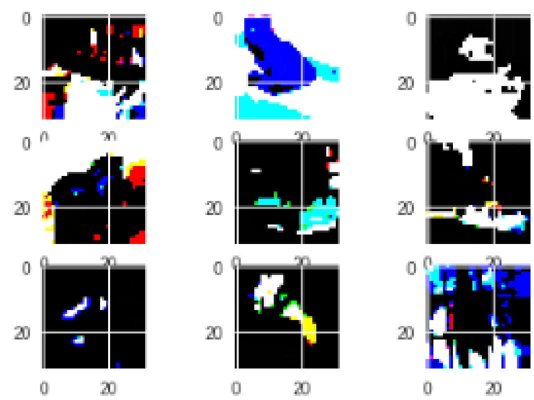
loss:



- examples of random chosen data:

data augmentation:

data normalization and augmentation:



3. We choose VGG16 as the pre-trained model and preform fine-tuning:

1. We take the first three blocks of the VGG16 model.
2. Flatten the output of layer block3_pool.
3. Add dense layer with 256 neurons with ‘relu’ activation function.
4. Add batch normalization and dropout
5. Add dense output layer with 10 neurons (that represent the class) with ‘softmax’ activation function because we want one output value- the class with maximal classification probability.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 32, 32, 3)         0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 32, 32, 64)        1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 32, 32, 64)        36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 16, 16, 64)        0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 16, 16, 128)       73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 16, 16, 128)       147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 8, 8, 128)         0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 8, 8, 256)         295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 8, 8, 256)         590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 8, 8, 256)         590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 4, 4, 256)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 4096)              0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               1048832
_________________________________________________________________
batch_normalization_1 (Batch (None, 256)               1024
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                2570
=================================================================
Total params: 2,787,914
Trainable params: 2,787,402
Non-trainable params: 512
```

<u>Notes:</u>

- the data is normalized with z-score normalization.
- We also want to note that without batch normalization layer the model didn’t learn- the train and validation accuracy have not changed during the epochs.

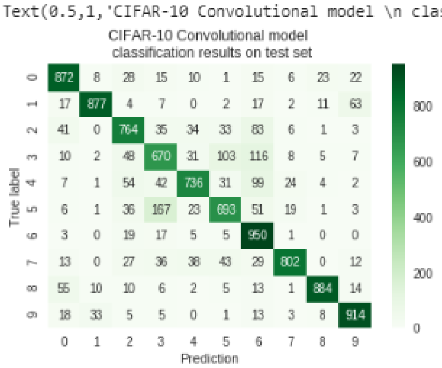
training process:

```
Train on 40000 samples, validate on 10000 samples
Epoch 1/5
40000/40000 [==============================] - 65s 2ms/step - loss: 1.3307 - acc: 0.5426 - val_loss: 4.5414 - val_acc: 0.1977
Epoch 2/5
40000/40000 [==============================] - 60s 2ms/step - loss: 0.9732 - acc: 0.6628 - val_loss: 4.1334 - val_acc: 0.2946
Epoch 3/5
40000/40000 [==============================] - 61s 2ms/step - loss: 0.7240 - acc: 0.7541 - val_loss: 0.6339 - val_acc: 0.7814
Epoch 4/5
40000/40000 [==============================] - 60s 2ms/step - loss: 0.6079 - acc: 0.7923 - val_loss: 0.5928 - val_acc: 0.7985
Epoch 5/5
40000/40000 [==============================] - 60s 2ms/step - loss: 0.5243 - acc: 0.8225 - val_loss: 0.5134 - val_acc: 0.8230
```

After 5 epochs we get ~ 82% test accuracy and ~87% train accuracy:

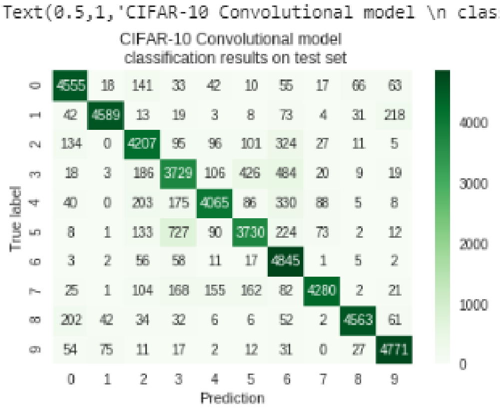
test accuracy:

```
model accuracy on test set is: 81.62%
Text(0.5,1,'CIFAR-10 Convolutional model \n classification results on test set')
```

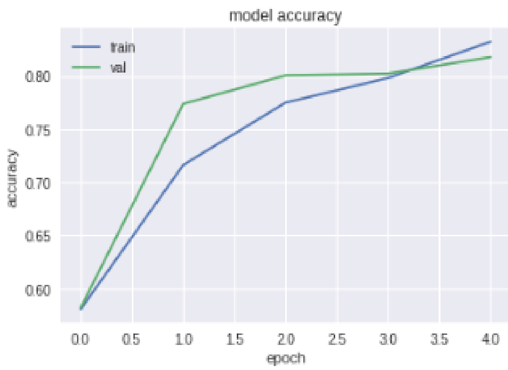


train accuracy:

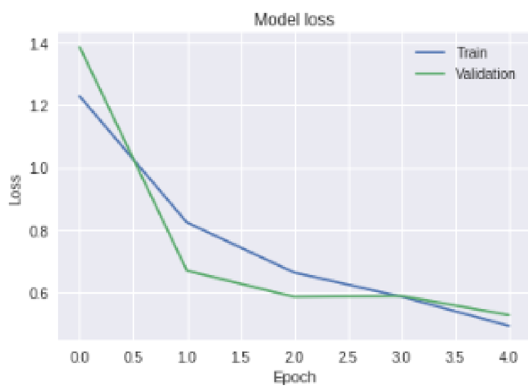
```
model accuracy on test set is: 86.668%
Text(0.5,1,'CIFAR-10 Convolutional model \n classification results on test set')
```



train-test accuracy:



loss:



After reading online blogs, we understand that the validation set accuracy is usually slightly better than train set accuracy because we add dropout layer with 0.5 dropout so the train process use less neurons then in validation and test process that go throw the entire network.

Now we tried to use the entire VGG16 network (all first five blocks and not only the first three):

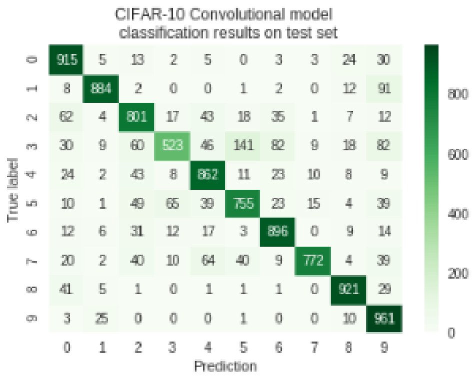
training process:

```
Train on 40000 samples, validate on 10000 samples
Epoch 1/5
40000/40000 [==============================] - 93s 2ms/step - loss: 1.0551 - acc: 0.6357 - val_loss: 0.7246 - val_acc: 0.7451
Epoch 2/5
40000/40000 [==============================] - 89s 2ms/step - loss: 0.5965 - acc: 0.8020 - val_loss: 0.7074 - val_acc: 0.7559
Epoch 3/5
40000/40000 [==============================] - 90s 2ms/step - loss: 0.4497 - acc: 0.8503 - val_loss: 0.4893 - val_acc: 0.8321
Epoch 4/5
40000/40000 [==============================] - 90s 2ms/step - loss: 0.3381 - acc: 0.8842 - val_loss: 0.5604 - val_acc: 0.8146
Epoch 5/5
40000/40000 [==============================] - 89s 2ms/step - loss: 0.2568 - acc: 0.9131 - val_loss: 0.5282 - val_acc: 0.8339
```

After 5 epochs we get ~ 83% test accuracy and ~91% train accuracy:

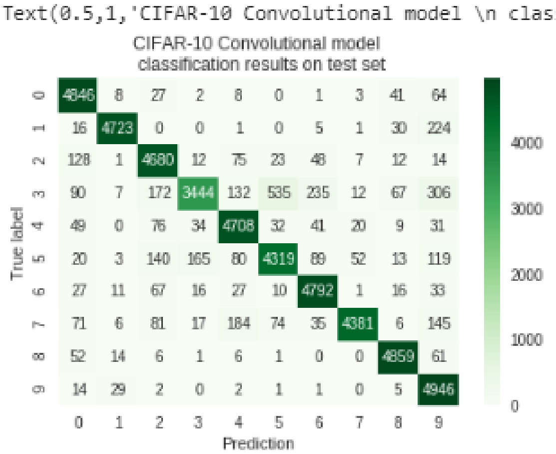
<u>test accuracy:</u>

```
model accuracy on test set is: 82.89999999999999%
Text(0.5,1,'CIFAR-10 Convolutional model \n classification results on test set')
```

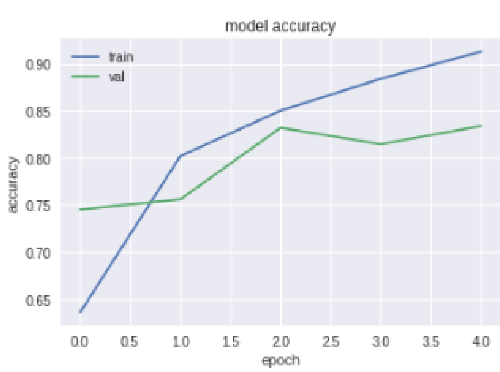


<u>train accuracy:</u>

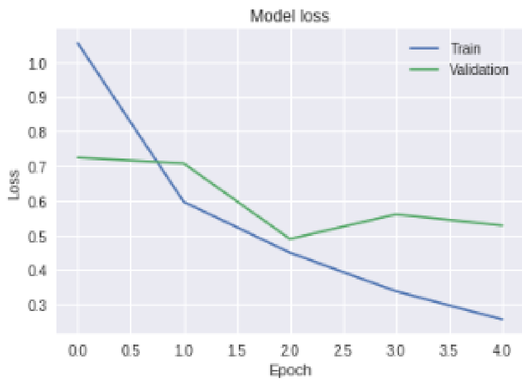
```
model accuracy on test set is: 91.396%
Text(0.5,1,'CIFAR-10 Convolutional model \n classification results on test set')
```



<u>train-test accuracy:</u>

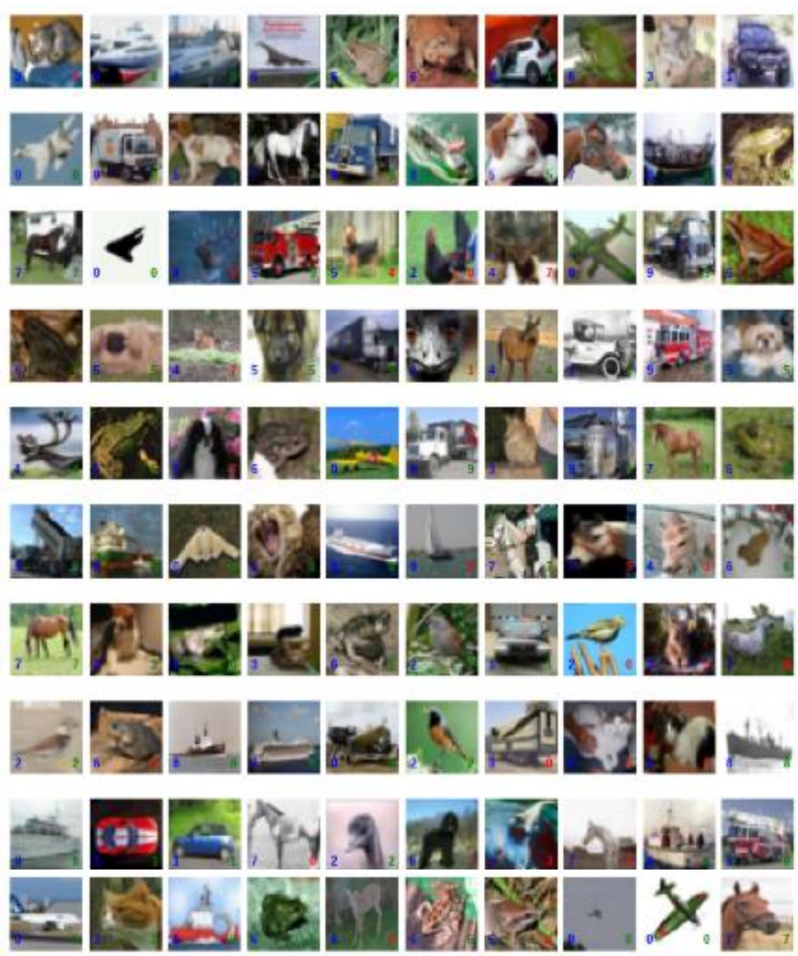


loos:



Example of good and bad classification:

Note: the class is sign with number, when blue number is the labeled class, green number is the right prediction and red number is wrong prediction.



Now, we tried to normalized the data using the mean and std of the original ImageNet data set, because the VGG16 model trained on this data. Therefore we want to pre-process the data by the orignal data:

After 5 epochs we get ~ 83% test accuracy and ~92% train accuracy (no significant improvement from the last run):

<u>normaliztion</u>:

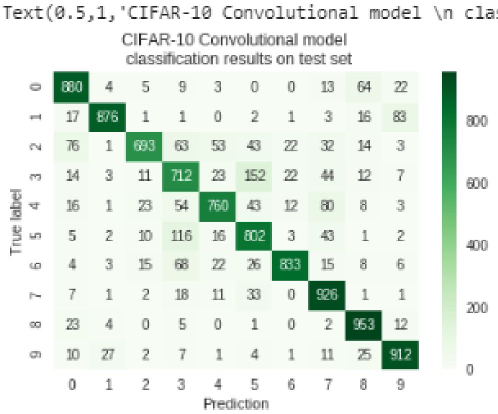
```
mean = [0.485, 0.456, 0.406]
std = [0.229, 0.224, 0.225]

def normalization(x):
  x = x.astype('float32')
  x = x/255
  x[..., 0] -= mean[0]
  x[..., 1] -= mean[1]
  x[..., 2] -= mean[2]
  x[..., 0] /= std[0]
  x[..., 1] /= std[1]
  x[..., 2] /= std[2]
  return x

x_train = normalization(x_train)
x_test = normalization(x_test)
```

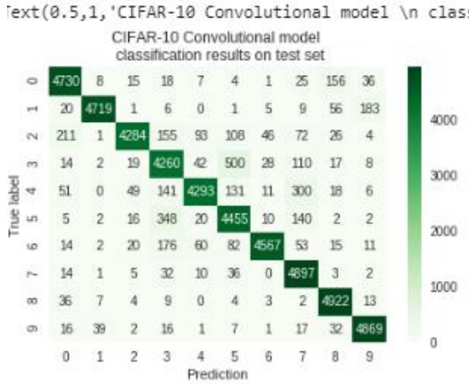
<u>test accuracy:</u>

```
model accuracy on test set is: 83.47%
Text(0.5,1,'CIFAR-10 Convolutional model \n classification results on test set')
```

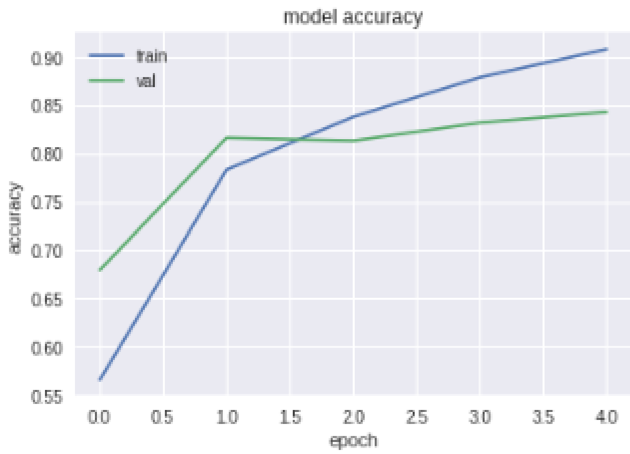


<u>train accuracy:</u>

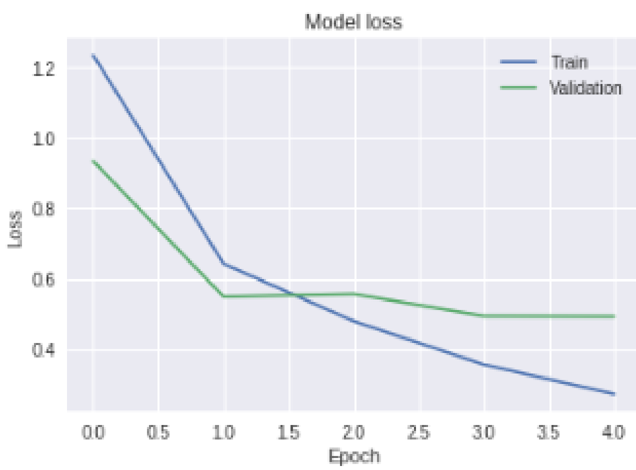
```
nodel accuracy on test set is: 91.99199999999999%
Text(0.5,1,'CIFAR-10 Convolutional model \n classification results on test set')
```



train-test accuracy:



loss:



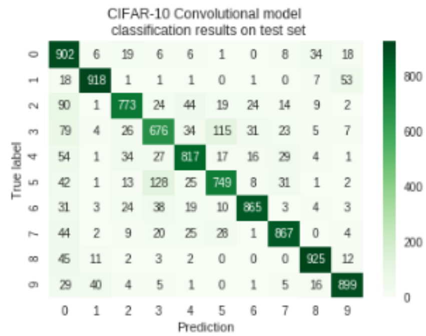
we assume that there is no significant improvement because the data of imagenet is very simmiler to CIFAR10 (mean and std values is simmiler). we save this model and used it as the feature extractore.

3.d we use the model we got in 3.c as a “feature extractor”:

1. we omit the last layer of the model
2. we get the prediction of this model on the train set
3. we use simple KNN classifier and train it with the output of the prediction of the model on the train set

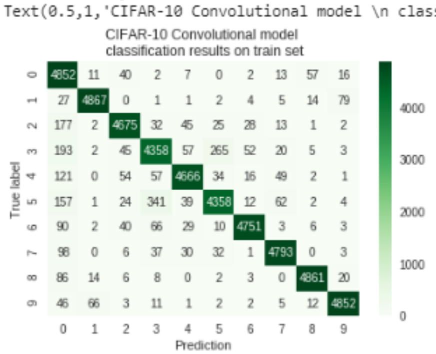
<u>KNN test accuracy:</u>

```
model accuracy on test set is: 83.91%
Text(0.5,1,'CIFAR-10 Convolutional model \n classification results on test set')
```



<u>KNN train accuracy:</u>

```
model accuracy on train set is: 94.066%
Text(0.5,1,'CIFAR-10 Convolutional model \n classification results on train set')
```



We can see that a simple classifier as KNN succeeded to achieve high accuracy on test ~84% and train set~94% only by using the output of the last layer of the CNN model as input for the training process of the KNN model.