Convolutional Neural Networks

This notebook explains CNNs through the implementation in sklearn.

We train a CNN to classify images from the CIFAR-10 database.

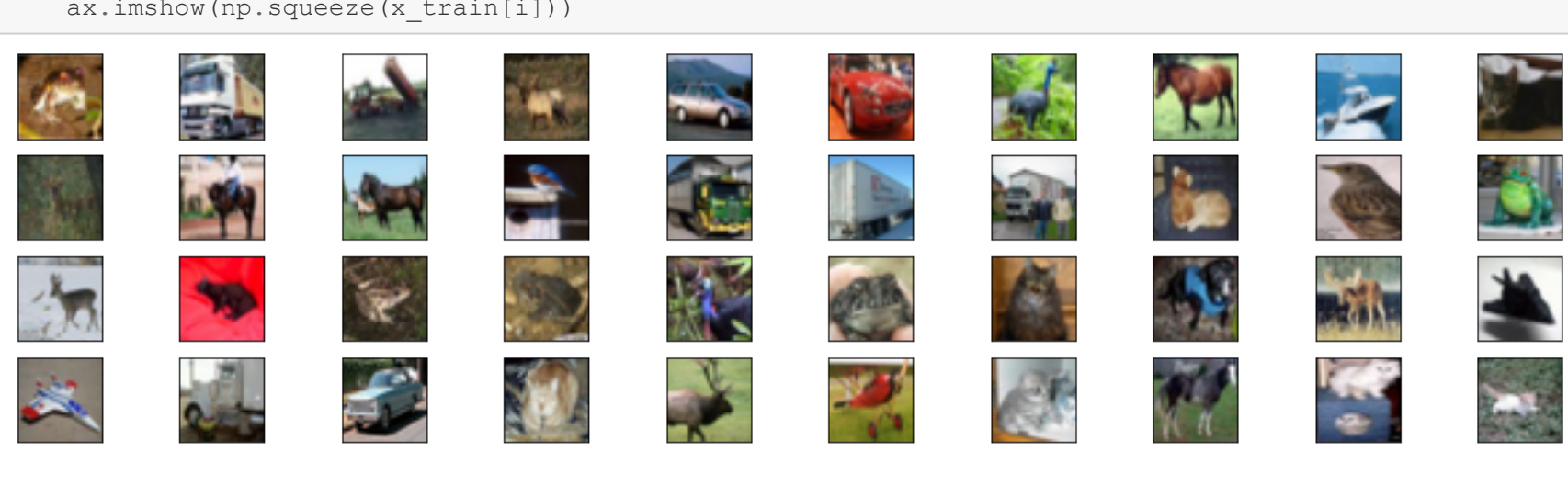
- The CIFAR-10 dataset consists of 60,000 colour images of size (32x32) in 10 classes, with 6000 images per class.
- Here are the classes in the dataset, as well as 10 random images from each:
  - airplane
  - automobile
  - bird
  - cat
  - deer
  - dog
  - frog
  - horse
  - ship
  - truck

1. Load CIFAR-10 Database

In [27]:
```
import keras
from keras.datasets import cifar10

# load the pre-shuffled train and test data
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

2. Visualize Some Training Images

In [28]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

fig = plt.figure(figsize=(20,5))
for i in range(40):
    ax = fig.add_subplot(4, 10, i + 1, xticks=[], yticks=[])
    ax.imshow(np.squeeze(x_train[i]))
```



3. Rescale the Images by Dividing Every Pixel in Every Image by 255

- Let's see how a datapoint looks like:

In [29]:
```
print(x_train[1])
```

```
[[[154 177 187]
  [126 137 136]
  [105 104  95]
  ...
  [ 91  95  71]
  [ 87  90  71]
  [ 79  81  70]]

 [[140 160 169]
  [145 153 154]
  [125 125 118]
  ...
  [ 96  99  78]
  [ 77  80  62]
  [ 71  73  61]]

 [[140 155 164]
  [139 146 149]
  [115 115 112]
  ...
  [ 79  82  64]
  [ 68  70  55]
  [ 67  69  55]]

 ...

 [[175 167 166]
  [156 154 160]
  [154 160 170]
  ...
  [ 42  34  36]
  [ 61  53  57]
  [ 93  83  91]]

 [[165 154 128]
  [156 152 130]
  [159 161 142]
  ...
  [103  93  96]
  [123 114 120]
  [131 121 131]]

 [[163 148 120]
  [158 148 122]
  [163 156 133]
  ...
  [143 133 139]
  [143 134 142]
  [143 133 144]]]
```

- Let's normalize each cell value [0,255], so that each cell value lies within the same distribution [0,1].
  - That is, the values change from [0,255] --> [0,1].
- We normalize using the formula:

$$\frac{X - X_{min}}{X_{max} - X_{min}}$$

- Here, X_min=0 and X_max=255.
- Hence,

$$\frac{X - X_{min}}{X_{max} - X_{min}} = \frac{X}{255}$$

In [30]:
```
x_train = x_train.astype('float32')/255
x_test = x_test.astype('float32')/255
```

4. Break Dataset into Training, Testing, and Validation Sets

- A datapoint in the dataset could belong to any one of the 10 classes mentioned earlier.
- Let's convert the output into one-hot encoded vector of size 10.
- This will be helpful for our CNN to recognize a given image datapoint into one of the 10 classes.

In [31]:
```
from keras.utils import np_utils

# one-hot encoding
num_classes = len(np.unique(y_train))
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

In [32]:
```
# break training set into training and validation sets
(x_train, x_valid) = x_train[5000:], x_train[:5000]
(y_train, y_valid) = y_train[5000:], y_train[:5000]

# print shape of training set
print('x_train shape:', x_train.shape)

# print number of training, validation, and test images
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
print(x_valid.shape[0], 'validation samples')
```

```
x_train shape: (45000, 32, 32, 3)
45000 train samples
10000 test samples
5000 validation samples
```

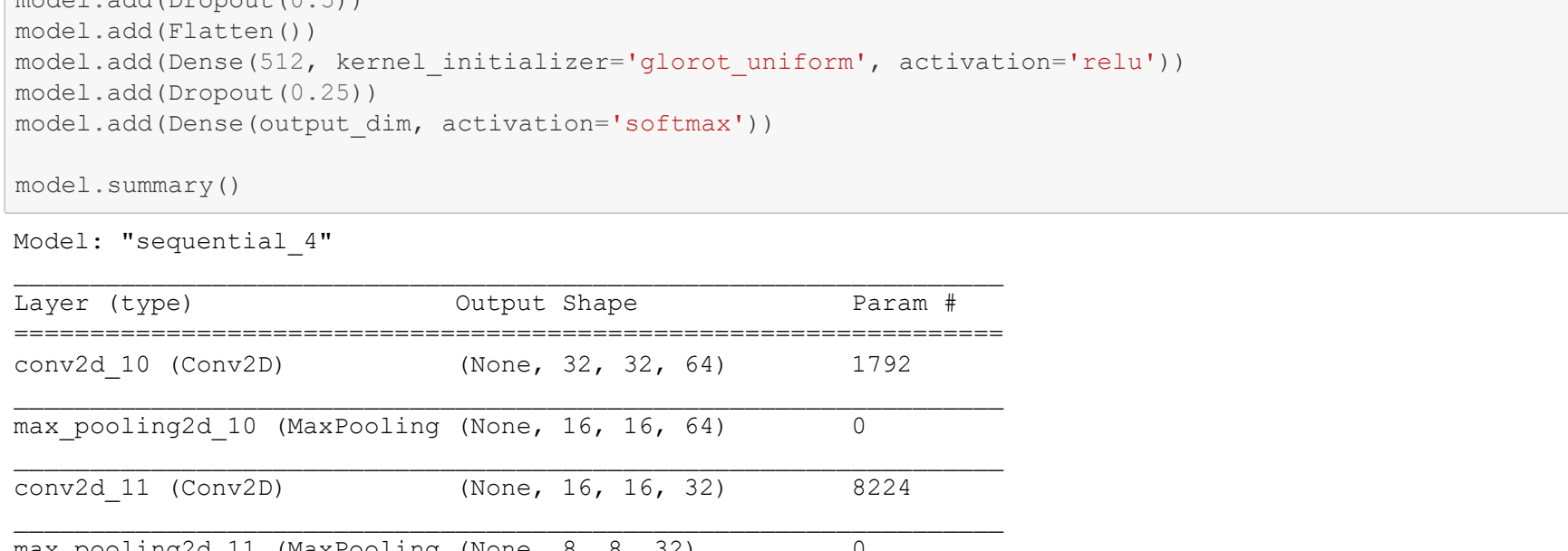
5. Data Augmentation

- Data augmentation helps in creating invariance in the data, especially in image data.
  - Invariance in data could be for:
    - Shift
    - Rotation
    - Zoom
    - Noise, etc

In [33]:
```
from keras.preprocessing.image import ImageDataGenerator

# create and configure augmented image generator
datagen_train = ImageDataGenerator(
    width_shift_range=0.1,  # randomly shift images horizontally (10% of total width)
    height_shift_range=0.1,  # randomly shift images vertically (10% of total height)
    horizontal_flip=True) # randomly flip images horizontally

# fit augmented image generator on data
datagen_train.fit(x_train)
```

6. Visualize Original and Augmented Images

In [34]:
```
import matplotlib.pyplot as plt

# take subset of training data
x_train_subset = x_train[:12]

# visualize subset of training data
fig = plt.figure(figsize=(20,2))
for i in range(0, len(x_train_subset)):
    ax = fig.add_subplot(1, 12, i+1)
    ax.imshow(x_train_subset[i])
fig.suptitle('Subset of Original Training Images', fontsize=20)
plt.show()

# visualize augmented images
fig = plt.figure(figsize=(20,2))
for x_batch in datagen_train.flow(x_train_subset, batch_size=12):
    for i in range(0, 12):
        ax = fig.add_subplot(1, 12, i+1)
        ax.imshow(x_batch[i])
    fig.suptitle('Augmented Images', fontsize=20)
    plt.show()
    break;
```



7. Define the Model Architecture

In [35]:
```
# Necessary model parameters:
output_dim = 10
input_shape = x_train[1].shape
batch_size = 30
epochs = 100
```

In [36]:
```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout


model = Sequential()
model.add(Conv2D(filters=64, kernel_size=3, padding='same', kernel_initializer='glorot_uniform',
                 activation='relu', input_shape=input_shape))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=32, kernel_size=2, padding='same', kernel_initializer='glorot_uniform',
                 activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.5))
model.add(Conv2D(filters=32, kernel_size=2, kernel_initializer='glorot_uniform', padding='same',
                 activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dense(512, kernel_initializer='glorot_uniform', activation='relu'))
model.add(Dropout(0.25))
model.add(Dense(output_dim, activation='softmax'))

model.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_10 (Conv2D)           (None, 32, 32, 64)        1792
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 16, 16, 64)        0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 16, 16, 32)        8224
_________________________________________________________________
max_pooling2d_11 (MaxPooling (None, 8, 8, 32)          0
_________________________________________________________________
dropout_10 (Dropout)         (None, 8, 8, 32)          0
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 8, 8, 32)          4128
_________________________________________________________________
max_pooling2d_12 (MaxPooling (None, 4, 4, 32)          0
_________________________________________________________________
dropout_11 (Dropout)         (None, 4, 4, 32)          0
_________________________________________________________________
flatten_4 (Flatten)          (None, 512)               0
_________________________________________________________________
dense_7 (Dense)              (None, 512)               262656
_________________________________________________________________
dropout_12 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_8 (Dense)              (None, 10)                5130
=================================================================
Total params: 281,930
Trainable params: 281,930
Non-trainable params: 0
_________________________________________________________________
```

8. Compile the Model

In [37]:
```
# compile the model
model.compile(loss='categorical_crossentropy', optimizer='Adam',
                  metrics=['accuracy'])
```

9. Train the Model

In [38]:
```
from keras.callbacks import ModelCheckpoint

# train the model
checkpointer = ModelCheckpoint(filepath='model.weights.best.hdf5', verbose=1,
                               save_best_only=True)
hist = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,
          validation_data=(x_valid, y_valid), callbacks=[checkpointer],
          verbose=2, shuffle=True)
```

```
Train on 45000 samples, validate on 5000 samples
Epoch 1/100
 - 15s - loss: 1.7699 - accuracy: 0.3441 - val_loss: 1.4642 - val_accuracy: 0.4850

Epoch 00001: val_loss improved from inf to 1.46422, saving model to model.weights.best.hdf5
Epoch 2/100
 - 15s - loss: 1.4993 - accuracy: 0.4504 - val_loss: 1.3215 - val_accuracy: 0.5300

Epoch 00002: val_loss improved from 1.46422 to 1.32148, saving model to model.weights.best.hdf5
Epoch 3/100
 - 15s - loss: 1.4107 - accuracy: 0.4857 - val_loss: 1.2507 - val_accuracy: 0.5604

Epoch 00003: val_loss improved from 1.32148 to 1.25072, saving model to model.weights.best.hdf5
Epoch 4/100
 - 15s - loss: 1.3623 - accuracy: 0.5058 - val_loss: 1.1976 - val_accuracy: 0.5858

Epoch 00004: val_loss improved from 1.25072 to 1.19757, saving model to model.weights.best.hdf5
Epoch 5/100
 - 16s - loss: 1.3085 - accuracy: 0.5284 - val_loss: 1.1358 - val_accuracy: 0.6094

Epoch 00005: val_loss improved from 1.19757 to 1.13583, saving model to model.weights.best.hdf5
Epoch 6/100
 - 15s - loss: 1.2796 - accuracy: 0.5359 - val_loss: 1.1268 - val_accuracy: 0.6048

Epoch 00006: val_loss improved from 1.13583 to 1.12677, saving model to model.weights.best.hdf5
Epoch 7/100
 - 15s - loss: 1.2493 - accuracy: 0.5507 - val_loss: 1.1171 - val_accuracy: 0.6106

Epoch 00007: val_loss improved from 1.12677 to 1.11715, saving model to model.weights.best.hdf5
Epoch 8/100
 - 15s - loss: 1.2303 - accuracy: 0.5562 - val_loss: 1.0861 - val_accuracy: 0.6306

Epoch 00008: val_loss improved from 1.11715 to 1.08607, saving model to model.weights.best.hdf5
Epoch 9/100
 - 15s - loss: 1.2055 - accuracy: 0.5691 - val_loss: 1.0619 - val_accuracy: 0.6426

Epoch 00009: val_loss improved from 1.08607 to 1.06191, saving model to model.weights.best.hdf5
Epoch 10/100
 - 15s - loss: 1.1855 - accuracy: 0.5734 - val_loss: 1.0209 - val_accuracy: 0.6498

Epoch 00010: val_loss improved from 1.06191 to 1.02088, saving model to model.weights.best.hdf5
Epoch 11/100
 - 15s - loss: 1.1649 - accuracy: 0.5828 - val_loss: 1.0192 - val_accuracy: 0.6540

Epoch 00011: val_loss improved from 1.02088 to 1.01922, saving model to model.weights.best.hdf5
Epoch 12/100
 - 15s - loss: 1.1558 - accuracy: 0.5875 - val_loss: 0.9984 - val_accuracy: 0.6636

Epoch 00012: val_loss improved from 1.01922 to 0.99839, saving model to model.weights.best.hdf5
Epoch 13/100
 - 15s - loss: 1.1404 - accuracy: 0.5926 - val_loss: 1.0733 - val_accuracy: 0.6348

Epoch 00013: val_loss did not improve from 0.99839
Epoch 14/100
 - 15s - loss: 1.1339 - accuracy: 0.5970 - val_loss: 0.9588 - val_accuracy: 0.6736

Epoch 00014: val_loss improved from 0.99839 to 0.95878, saving model to model.weights.best.hdf5
Epoch 15/100
 - 15s - loss: 1.1161 - accuracy: 0.5995 - val_loss: 0.9537 - val_accuracy: 0.6636

Epoch 00015: val_loss improved from 0.95878 to 0.95370, saving model to model.weights.best.hdf5
Epoch 16/100
 - 15s - loss: 1.1044 - accuracy: 0.6055 - val_loss: 0.9636 - val_accuracy: 0.6724

Epoch 00016: val_loss did not improve from 0.95370
Epoch 17/100
 - 15s - loss: 1.0963 - accuracy: 0.6110 - val_loss: 0.9344 - val_accuracy: 0.6722

Epoch 00017: val_loss improved from 0.95370 to 0.93436, saving model to model.weights.best.hdf5
Epoch 18/100
 - 15s - loss: 1.0893 - accuracy: 0.6108 - val_loss: 0.9407 - val_accuracy: 0.6758

Epoch 00018: val_loss did not improve from 0.93436
Epoch 19/100
 - 15s - loss: 1.0830 - accuracy: 0.6126 - val_loss: 0.9042 - val_accuracy: 0.6972

Epoch 00019: val_loss improved from 0.93436 to 0.90416, saving model to model.weights.best.hdf5
Epoch 20/100
 - 15s - loss: 1.0723 - accuracy: 0.6182 - val_loss: 0.9095 - val_accuracy: 0.6880

Epoch 00020: val_loss did not improve from 0.90416
Epoch 21/100
 - 15s - loss: 1.0640 - accuracy: 0.6202 - val_loss: 0.9240 - val_accuracy: 0.6886

Epoch 00021: val_loss did not improve from 0.90416
Epoch 22/100
 - 15s - loss: 1.0563 - accuracy: 0.6226 - val_loss: 0.9013 - val_accuracy: 0.6946

Epoch 00022: val_loss improved from 0.90416 to 0.90126, saving model to model.weights.best.hdf5
Epoch 23/100
 - 15s - loss: 1.0523 - accuracy: 0.6220 - val_loss: 0.8850 - val_accuracy: 0.7118

Epoch 00023: val_loss improved from 0.90126 to 0.88497, saving model to model.weights.best.hdf5
Epoch 24/100
 - 15s - loss: 1.0484 - accuracy: 0.6268 - val_loss: 0.8535 - val_accuracy: 0.7204

Epoch 00024: val_loss improved from 0.88497 to 0.85346, saving model to model.weights.best.hdf5
Epoch 25/100
 - 16s - loss: 1.0414 - accuracy: 0.6288 - val_loss: 0.8651 - val_accuracy: 0.7080

Epoch 00025: val_loss did not improve from 0.85346
Epoch 26/100
 - 15s - loss: 1.0329 - accuracy: 0.6333 - val_loss: 0.9203 - val_accuracy: 0.6950

Epoch 00026: val_loss did not improve from 0.85346
Epoch 27/100
 - 15s - loss: 1.0339 - accuracy: 0.6316 - val_loss: 0.9300 - val_accuracy: 0.6844

Epoch 00027: val_loss did not improve from 0.85346
Epoch 28/100
 - 15s - loss: 1.0235 - accuracy: 0.6342 - val_loss: 0.9007 - val_accuracy: 0.6886

Epoch 00028: val_loss did not improve from 0.85346
Epoch 29/100
 - 15s - loss: 1.0260 - accuracy: 0.6352 - val_loss: 0.8611 - val_accuracy: 0.7102

Epoch 00029: val_loss did not improve from 0.85346
Epoch 30/100
 - 15s - loss: 1.0194 - accuracy: 0.6383 - val_loss: 0.8520 - val_accuracy: 0.7074

Epoch 00030: val_loss improved from 0.85346 to 0.85199, saving model to model.weights.best.hdf5
Epoch 31/100
 - 15s - loss: 1.0066 - accuracy: 0.6416 - val_loss: 0.8477 - val_accuracy: 0.7200

Epoch 00031: val_loss improved from 0.85199 to 0.84770, saving model to model.weights.best.hdf5
Epoch 32/100
 - 15s - loss: 1.0061 - accuracy: 0.6430 - val_loss: 0.8421 - val_accuracy: 0.7138

Epoch 00032: val_loss improved from 0.84770 to 0.84205, saving model to model.weights.best.hdf5
Epoch 33/100
 - 15s - loss: 1.0001 - accuracy: 0.6456 - val_loss: 0.8314 - val_accuracy: 0.7298

Epoch 00033: val_loss improved from 0.84205 to 0.83141, saving model to model.weights.best.hdf5
Epoch 34/100
 - 15s - loss: 0.9941 - accuracy: 0.6470 - val_loss: 0.8352 - val_accuracy: 0.7150

Epoch 00034: val_loss did not improve from 0.83141
Epoch 35/100
 - 15s - loss: 0.9940 - accuracy: 0.6471 - val_loss: 0.8256 - val_accuracy: 0.7218

Epoch 00035: val_loss improved from 0.83141 to 0.82561, saving model to model.weights.best.hdf5
Epoch 36/100
 - 15s - loss: 0.9931 - accuracy: 0.6458 - val_loss: 0.8203 - val_accuracy: 0.7286

Epoch 00036: val_loss improved from 0.82561 to 0.82029, saving model to model.weights.best.hdf5
Epoch 37/100
 - 15s - loss: 0.9868 - accuracy: 0.6503 - val_loss: 0.8679 - val_accuracy: 0.6992

Epoch 00037: val_loss did not improve from 0.82029
Epoch 38/100
 - 15s - loss: 0.9885 - accuracy: 0.6489 - val_loss: 0.7887 - val_accuracy: 0.7384

Epoch 00038: val_loss improved from 0.82029 to 0.78866, saving model to model.weights.best.hdf5
Epoch 39/100
 - 15s - loss: 0.9778 - accuracy: 0.6521 - val_loss: 0.8310 - val_accuracy: 0.7198

Epoch 00039: val_loss did not improve from 0.78866
Epoch 40/100
 - 15s - loss: 0.9789 - accuracy: 0.6536 - val_loss: 0.7944 - val_accuracy: 0.7346

Epoch 00040: val_loss did not improve from 0.78866
Epoch 41/100
 - 15s - loss: 0.9757 - accuracy: 0.6544 - val_loss: 0.8277 - val_accuracy: 0.7292

Epoch 00041: val_loss did not improve from 0.78866
Epoch 42/100
 - 15s - loss: 0.9750 - accuracy: 0.6538 - val_loss: 0.8148 - val_accuracy: 0.7294

Epoch 00042: val_loss did not improve from 0.78866
Epoch 43/100
 - 15s - loss: 0.9688 - accuracy: 0.6592 - val_loss: 0.7873 - val_accuracy: 0.7388

Epoch 00043: val_loss improved from 0.78866 to 0.78730, saving model to model.weights.best.hdf5
Epoch 44/100
 - 15s - loss: 0.9636 - accuracy: 0.6585 - val_loss: 0.8219 - val_accuracy: 0.7274

Epoch 00044: val_loss did not improve from 0.78730
Epoch 45/100
 - 15s - loss: 0.9569 - accuracy: 0.6607 - val_loss: 0.8037 - val_accuracy: 0.7348

Epoch 00045: val_loss did not improve from 0.78730
Epoch 46/100
 - 16s - loss: 0.9562 - accuracy: 0.6612 - val_loss: 0.8097 - val_accuracy: 0.7332

Epoch 00046: val_loss did not improve from 0.78730
Epoch 47/100
 - 15s - loss: 0.9557 - accuracy: 0.6602 - val_loss: 0.7919 - val_accuracy: 0.7386

Epoch 00047: val_loss did not improve from 0.78730
Epoch 48/100
 - 15s - loss: 0.9581 - accuracy: 0.6583 - val_loss: 0.7752 - val_accuracy: 0.7442

Epoch 00048: val_loss improved from 0.78730 to 0.77517, saving model to model.weights.best.hdf5
Epoch 49/100
 - 15s - loss: 0.9554 - accuracy: 0.6630 - val_loss: 0.7976 - val_accuracy: 0.7386

Epoch 00049: val_loss did not improve from 0.77517
Epoch 50/100
 - 15s - loss: 0.9522 - accuracy: 0.6635 - val_loss: 0.8018 - val_accuracy: 0.7258

Epoch 00050: val_loss did not improve from 0.77517
Epoch 51/100
 - 15s - loss: 0.9528 - accuracy: 0.6648 - val_loss: 0.7826 - val_accuracy: 0.7360

Epoch 00051: val_loss did not improve from 0.77517
Epoch 52/100
 - 15s - loss: 0.9503 - accuracy: 0.6614 - val_loss: 0.8125 - val_accuracy: 0.7260

Epoch 00052: val_loss did not improve from 0.77517
Epoch 53/100
 - 15s - loss: 0.9453 - accuracy: 0.6678 - val_loss: 0.7805 - val_accuracy: 0.7364

Epoch 00053: val_loss did not improve from 0.77517
Epoch 54/100
 - 15s - loss: 0.9470 - accuracy: 0.6645 - val_loss: 0.7783 - val_accuracy: 0.7396

Epoch 00054: val_loss did not improve from 0.77517
Epoch 55/100
 - 15s - loss: 0.9504 - accuracy: 0.6612 - val_loss: 0.8476 - val_accuracy: 0.7076

Epoch 00055: val_loss did not improve from 0.77517
Epoch 56/100
 - 15s - loss: 0.9426 - accuracy: 0.6667 - val_loss: 0.8012 - val_accuracy: 0.7304

Epoch 00056: val_loss did not improve from 0.77517
Epoch 57/100
 - 15s - loss: 0.9373 - accuracy: 0.6677 - val_loss: 0.7725 - val_accuracy: 0.7418

Epoch 00057: val_loss improved from 0.77517 to 0.77248, saving model to model.weights.best.hdf5
Epoch 58/100
 - 15s - loss: 0.9489 - accuracy: 0.6649 - val_loss: 0.7760 - val_accuracy: 0.7492

Epoch 00058: val_loss did not improve from 0.77248
Epoch 59/100
 - 15s - loss: 0.9386 - accuracy: 0.6679 - val_loss: 0.7761 - val_accuracy: 0.7412

Epoch 00059: val_loss did not improve from 0.77248
Epoch 60/100
 - 15s - loss: 0.9370 - accuracy: 0.6704 - val_loss: 0.7905 - val_accuracy: 0.7316

Epoch 00060: val_loss did not improve from 0.77248
Epoch 61/100
 - 15s - loss: 0.9339 - accuracy: 0.6708 - val_loss: 0.7938 - val_accuracy: 0.7382

Epoch 00061: val_loss did not improve from 0.77248
Epoch 62/100
 - 15s - loss: 0.9325 - accuracy: 0.6685 - val_loss: 0.7621 - val_accuracy: 0.7374

Epoch 00062: val_loss improved from 0.77248 to 0.76211, saving model to model.weights.best.hdf5
Epoch 63/100
 - 15s - loss: 0.9333 - accuracy: 0.6703 - val_loss: 0.7544 - val_accuracy: 0.7450

Epoch 00063: val_loss improved from 0.76211 to 0.75443, saving model to model.weights.best.hdf5
Epoch 64/100
 - 15s - loss: 0.9299 - accuracy: 0.6724 - val_loss: 0.7948 - val_accuracy: 0.7320

Epoch 00064: val_loss did not improve from 0.75443
Epoch 65/100
 - 15s - loss: 0.9320 - accuracy: 0.6704 - val_loss: 0.7691 - val_accuracy: 0.7384

Epoch 00065: val_loss did not improve from 0.75443
Epoch 66/100
 - 15s - loss: 0.9304 - accuracy: 0.6705 - val_loss: 0.7641 - val_accuracy: 0.7416

Epoch 00066: val_loss did not improve from 0.75443
Epoch 67/100
 - 15s - loss: 0.9255 - accuracy: 0.6724 - val_loss: 0.7989 - val_accuracy: 0.7300

Epoch 00067: val_loss did not improve from 0.75443
Epoch 68/100
 - 15s - loss: 0.9234 - accuracy: 0.6725 - val_loss: 0.7773 - val_accuracy: 0.7314

Epoch 00068: val_loss did not improve from 0.75443
Epoch 69/100
 - 15s - loss: 0.9130 - accuracy: 0.6760 - val_loss: 0.7832 - val_accuracy: 0.7328

Epoch 00069: val_loss did not improve from 0.75443
Epoch 70/100
 - 15s - loss: 0.9191 - accuracy: 0.6776 - val_loss: 0.7564 - val_accuracy: 0.7416

Epoch 00070: val_loss did not improve from 0.75443
Epoch 71/100
 - 15s - loss: 0.9220 - accuracy: 0.6743 - val_loss: 0.7744 - val_accuracy: 0.7342

Epoch 00071: val_loss did not improve from 0.75443
Epoch 72/100
 - 15s - loss: 0.9202 - accuracy: 0.6761 - val_loss: 0.7836 - val_accuracy: 0.7378

Epoch 00072: val_loss did not improve from 0.75443
Epoch 73/100
 - 15s - loss: 0.9149 - accuracy: 0.6777 - val_loss: 0.7749 - val_accuracy: 0.7348

Epoch 00073: val_loss did not improve from 0.75443
Epoch 74/100
 - 15s - loss: 0.9208 - accuracy: 0.6737 - val_loss: 0.7489 - val_accuracy: 0.7404

Epoch 00074: val_loss improved from 0.75443 to 0.74891, saving model to model.weights.best.hdf5
Epoch 75/100
 - 15s - loss: 0.9192 - accuracy: 0.6749 - val_loss: 0.8012 - val_accuracy: 0.7272

Epoch 00075: val_loss did not improve from 0.74891
Epoch 76/100
 - 15s - loss: 0.9163 - accuracy: 0.6761 - val_loss: 0.8335 - val_accuracy: 0.7138

Epoch 00076: val_loss did not improve from 0.74891
Epoch 77/100
 - 15s - loss: 0.9141 - accuracy: 0.6752 - val_loss: 0.7490 - val_accuracy: 0.7530

Epoch 00077: val_loss did not improve from 0.74891
Epoch 78/100
 - 15s - loss: 0.9149 - accuracy: 0.6770 - val_loss: 0.7663 - val_accuracy: 0.7420

Epoch 00078: val_loss did not improve from 0.74891
Epoch 79/100
 - 15s - loss: 0.9109 - accuracy: 0.6788 - val_loss: 0.7487 - val_accuracy: 0.7496

Epoch 00079: val_loss improved from 0.74891 to 0.74867, saving model to model.weights.best.hdf5
Epoch 80/100
 - 15s - loss: 0.9076 - accuracy: 0.6806 - val_loss: 0.8131 - val_accuracy: 0.7250

Epoch 00080: val_loss did not improve from 0.74867
Epoch 81/100
 - 15s - loss: 0.9087 - accuracy: 0.6787 - val_loss: 0.7461 - val_accuracy: 0.7486

Epoch 00081: val_loss improved from 0.74867 to 0.74614, saving model to model.weights.best.hdf5
Epoch 82/100
 - 15s - loss: 0.9033 - accuracy: 0.6798 - val_loss: 0.7749 - val_accuracy: 0.7440

Epoch 00082: val_loss did not improve from 0.74614
Epoch 83/100
 - 15s - loss: 0.9037 - accuracy: 0.6817 - val_loss: 0.7356 - val_accuracy: 0.7494

Epoch 00083: val_loss improved from 0.74614 to 0.73564, saving model to model.weights.best.hdf5
Epoch 84/100
 - 15s - loss: 0.9070 - accuracy: 0.6794 - val_loss: 0.7747 - val_accuracy: 0.7416

Epoch 00084: val_loss did not improve from 0.73564
Epoch 85/100
 - 15s - loss: 0.9083 - accuracy: 0.6787 - val_loss: 0.7298 - val_accuracy: 0.7598

Epoch 00085: val_loss improved from 0.73564 to 0.72983, saving model to model.weights.best.hdf5
Epoch 86/100
 - 15s - loss: 0.9114 - accuracy: 0.6767 - val_loss: 0.7446 - val_accuracy: 0.7536

Epoch 00086: val_loss did not improve from 0.72983
Epoch 87/100
 - 16s - loss: 0.9031 - accuracy: 0.6809 - val_loss: 0.7392 - val_accuracy: 0.7502

Epoch 00087: val_loss did not improve from 0.72983
Epoch 88/100
 - 15s - loss: 0.9086 - accuracy: 0.6805 - val_loss: 0.7148 - val_accuracy: 0.7630

Epoch 00088: val_loss improved from 0.72983 to 0.71484, saving model to model.weights.best.hdf5
Epoch 89/100
 - 15s - loss: 0.9098 - accuracy: 0.6787 - val_loss: 0.7368 - val_accuracy: 0.7514

Epoch 00089: val_loss did not improve from 0.71484
Epoch 90/100
 - 15s - loss: 0.9005 - accuracy: 0.6806 - val_loss: 0.7359 - val_accuracy: 0.7540

Epoch 00090: val_loss did not improve from 0.71484
Epoch 91/100
 - 15s - loss: 0.9006 - accuracy: 0.6794 - val_loss: 0.7393 - val_accuracy: 0.7526

Epoch 00091: val_loss did not improve from 0.71484
Epoch 92/100
 - 15s - loss: 0.8980 - accuracy: 0.6805 - val_loss: 0.7513 - val_accuracy: 0.7500

Epoch 00092: val_loss did not improve from 0.71484
Epoch 93/100
 - 15s - loss: 0.9088 - accuracy: 0.6786 - val_loss: 0.7682 - val_accuracy: 0.7398

Epoch 00093: val_loss did not improve from 0.71484
Epoch 94/100
 - 15s - loss: 0.9012 - accuracy: 0.6793 - val_loss: 0.7493 - val_accuracy: 0.7440

Epoch 00094: val_loss did not improve from 0.71484
Epoch 95/100
 - 15s - loss: 0.8963 - accuracy: 0.6844 - val_loss: 0.7441 - val_accuracy: 0.7480

Epoch 00095: val_loss did not improve from 0.71484
Epoch 96/100
 - 15s - loss: 0.8972 - accuracy: 0.6824 - val_loss: 0.7383 - val_accuracy: 0.7382

Epoch 00096: val_loss did not improve from 0.71484
Epoch 97/100
 - 15s - loss: 0.8959 - accuracy: 0.6854 - val_loss: 0.7501 - val_accuracy: 0.7510

Epoch 00097: val_loss did not improve from 0.71484
Epoch 98/100
 - 15s - loss: 0.9025 - accuracy: 0.6821 - val_loss: 0.7452 - val_accuracy: 0.7462

Epoch 00098: val_loss did not improve from 0.71484
Epoch 99/100
 - 15s - loss: 0.8956 - accuracy: 0.6862 - val_loss: 0.7508 - val_accuracy: 0.7510

Epoch 00099: val_loss did not improve from 0.71484
Epoch 100/100
 - 15s - loss: 0.8925 - accuracy: 0.6837 - val_loss: 0.7419 - val_accuracy: 0.7416

Epoch 00100: val_loss did not improve from 0.71484
```

Explaining each line of code:

- We start off by defininig some necessary model parameters such as:
  - input_dim: Defines the input layer size.
    - Preferred to be equal to the input datapoint size.
  - output_dim: Defines the output layer size.
    - Should to be equal to the output size.
  - batch_size: No. of datapoints taken in each batch, during the training process.
    - Why do we use batches of datapoints instead of all the datapoints at once for training?
      - Because:
        - It requires less memory.
        - Typically networks train faster with mini-batches.
          - That's because the weights update after each epoch.
  - epochs: No. of epochs(iterations over all the datapoints) for training.
- Next, we define our model to be Sequetial().
  - An artificial neural network is a sequential model.
    - The Sequential model is a linear stack of layers.
- add() method is used to add a layer into the sequential model.
- Conv2D creates a Convolutional layer for spatial 2d data
  - filters = no. of filters
  - kernel_size = size of each filter
  - padding:
    - 'valid': No padding
    - 'same': Same sized zero-padding
  - Note: A colored image is a 3d-tensor of 2d-images(Red, Green, and Blue)
- Activation function used in each layer is: ReLu
- Weights are initialized using: Glorot Uniform Distribution
- MaxPooling2D creates a Max-Pooling layer for 2d data:
  - Helps creating location-invariance in the image data.
    - Location-invariance is used to detect the subject in an image irrespective of its location in the image.
- Dropout is used to dropping out a few number of neurons in a layer, which results in regularising the MLP.
  - A certain number of neurons turn off at random, hence their outputs are not used in the model building.
  - Regularization is used for 'Bias-Variance Trade-offs', which refers to 'Overfitting' or 'Underfitting' a model.
- Flatten operation flattens the multidimensional data into 1-d array.
- 'Dense' implements the operation: output = activation(dot(input, kernel) + bias)
  - Where:
    - activation() is the element-wise activation function passed as the activation argument,
    - kernel is a weights matrix created by the layer,
    - bias is a bias vector created by the layer (only applicable if use_bias is True).
  - output = activation(dot(input, kernel) + bias)
    - y = activation(WT. X + b)
- Since, the CNN model built is supposed to recognise the images from 10 classes mentioned earlier, which is basically a 10-class classification task:
  - Hence, we use Softmax activation function in the output layer.
- summary() method prints the number of weight parameters and bias(if use_bias is True) parameters calculated in each layer.
- compile() method configures the learning process before training a model.
  - For optimizing the gradients, Adam optimizer is used.
- Keras models are trained on Numpy arrays of input data and labels.
- ModelCheckpoint is used to save the best weights yeilded by the model.
- For training a model, we typically use the fit() method.
  - fit() method trains the model for a fixed number of epochs (iterations on a dataset).
    - It returns a History object.
    - History attribute is a record of training loss values and metrics values at successive epochs, as well as validation loss values and validation metrics values (if applicable).
  - Verbose:
    - verbose=0 shows nothing (silent)
    - verbose=1 shows an animated progress bar like this:
      - progres_bar: [==============>]
    - verbose=2 mentions the number of epoch like this:
      - Epoch = 1/10
- Finally, we evaluate the model on the test data via evaluate().

10. Load the Model with the Best Validation Accuracy

In [39]:
```
# load the weights that yielded the best validation accuracy
model.load_weights('model.weights.best.hdf5')
```

11. Evaluating the Model on Test dataset

In [40]:
```
# evaluate and print test accuracy
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Test loss: 0.7480328998565674
Test accuracy: 0.7462000250816345
```

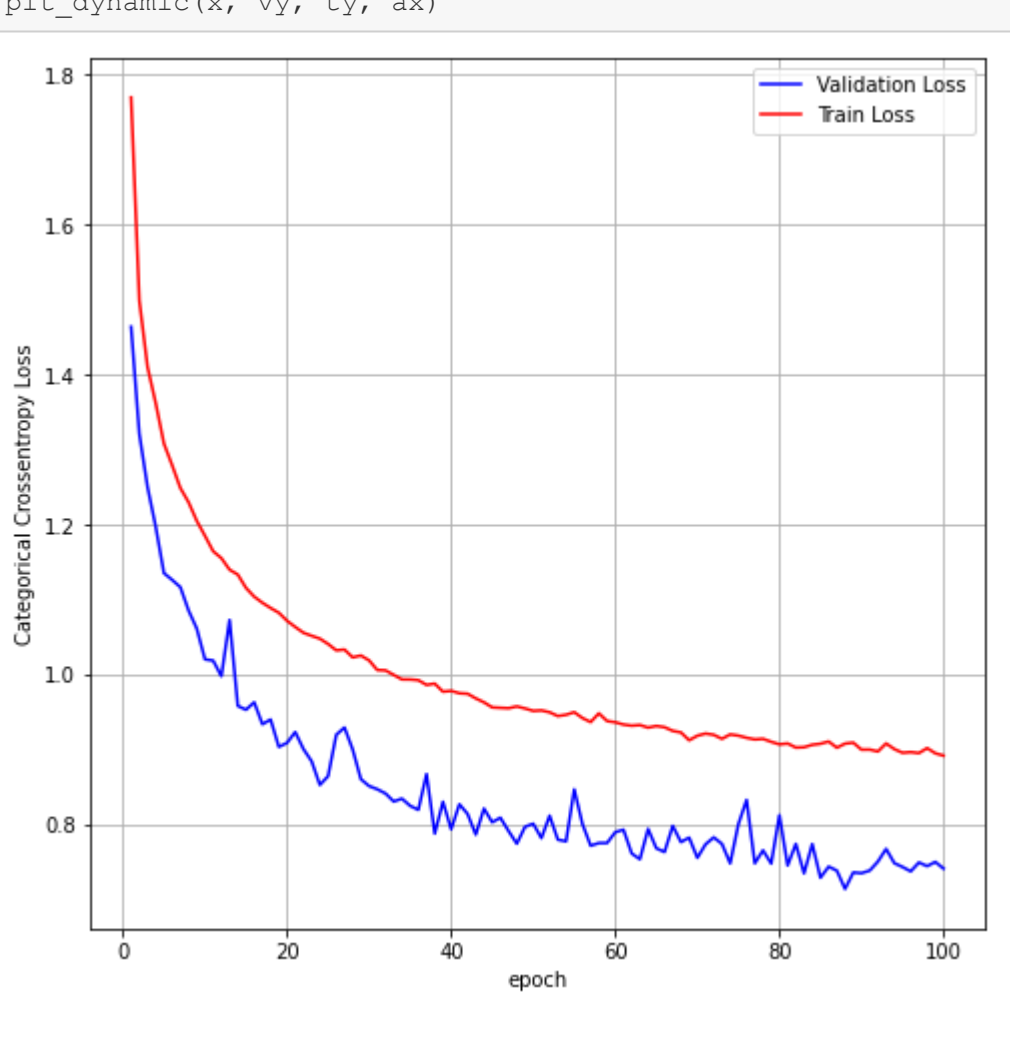
How to interpret loss and accuracy?

- Loss can be seen as a distance between the true values of the problem and the values predicted by the model.
  - Greater the loss is, more huge is the errors made on the data.
- Accuracy can be seen as the number of errors made on the data.

  That means :
  - a low accuracy and huge loss means:
    - the model made huge errors on a lot of data
  - a low accuracy but low loss means:
    - the model made little errors on a lot of data
  - a great accuracy with low loss means:
    - the model made low errors on a few data (best case)

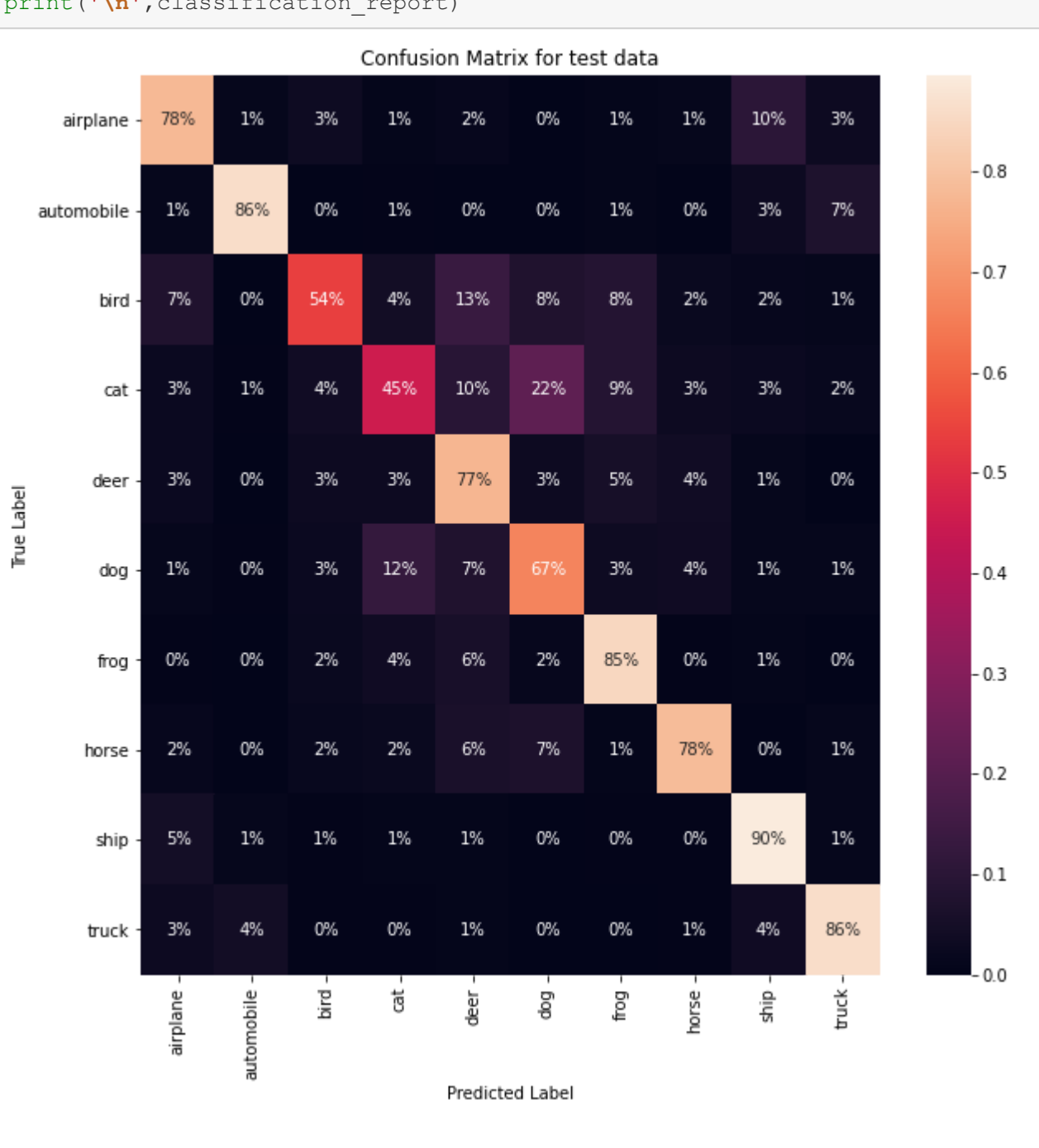
- The function below is used to update the plots for each epoch and error.

In [41]:
```
# https://gist.github.com/greydanus/f6eee59eaf1d90fcb3b534a25362cea4

def plt_dynamic(x, vy, ty, ax, colors=['b']):
  fig = plt.figure(figsize=(8,8))
  ax = plt.subplot(111)
  ax.plot(x, vy, 'b', label="Validation Loss")
  ax.plot(x, ty, 'r', label="Train Loss")
  ax.set_xlabel('epoch') ; ax.set_ylabel('Categorical Crossentropy Loss')
  ax.legend()
  plt.grid()
  fig.canvas.draw()

# list of epoch numbers
x = list(range(1,epochs+1))

vy = hist.history['val_loss']
ty = hist.history['loss']
plt_dynamic(x, vy, ty, ax)
```



- Confusion Matrix and Classification Report:

In [42]:
```
predic= model.predict(x_test)
predic= np.argmax(predic, axis=1)
y_test_new=np.argmax(y_test, axis=1)

class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'
]


#from sklearn.metrics import confusion_matrix
import pandas as pd
from sklearn import metrics
import seaborn as sns

# Confusion Matrix
conf_mat = metrics.confusion_matrix(y_test_new, predic)
conf_mat = conf_mat.astype('float') / conf_mat.sum(axis=1)[:, np.newaxis]

df = pd.DataFrame(conf_mat, index = class_names, columns = class_names)

# Plotting Confusion Matrix
plt.figure(figsize=(10,10))
sns.heatmap(df, annot=True, fmt='.0%')
plt.title("Confusion Matrix for test data")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()

# Classification Report
classification_report = metrics.classification_report(y_test_new, predic, target_names=class_names)

print('\n')
print('-'*20,'Metric report for test data','-'*20)
print('\n',classification_report)
```



True \ Predicted	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	78%	1%	3%	1%	2%	0%	1%	1%	10%	3%
automobile	1%	86%	0%	1%	0%	0%	1%	0%	3%	7%
bird	7%	0%	54%	4%	13%	8%	8%	2%	2%	1%
cat	3%	1%	4%	45%	10%	22%	9%	3%	3%	2%
deer	3%	0%	3%	3%	77%	3%	5%	4%	1%	0%
dog	1%	0%	3%	12%	7%	67%	3%	4%	1%	1%
frog	0%	0%	2%	4%	6%	2%	85%	0%	1%	0%
horse	2%	0%	2%	2%	6%	7%	1%	78%	0%	1%
ship	5%	1%	1%	1%	1%	0%	0%	0%	90%	1%
truck	3%	4%	0%	0%	1%	0%	0%	1%	4%	86%

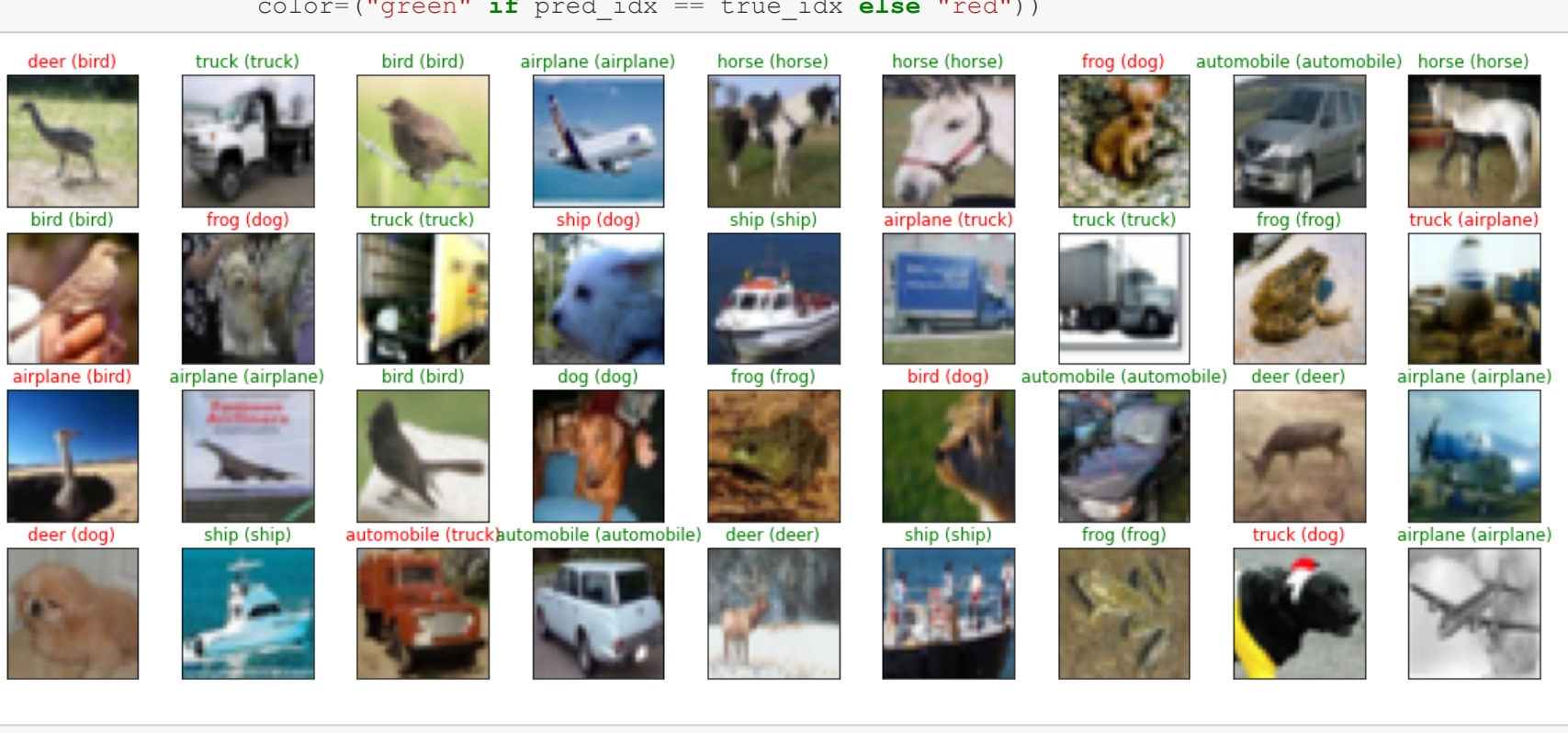
```
-------------------- Metric report for test data --------------------

              precision    recall  f1-score   support

    airplane       0.76      0.78      0.77      1000
  automobile       0.91      0.86      0.89      1000
        bird       0.74      0.54      0.62      1000
         cat       0.61      0.45      0.52      1000
        deer       0.63      0.77      0.69      1000
         dog       0.61      0.67      0.64      1000
        frog       0.75      0.85      0.80      1000
       horse       0.83      0.79      0.81      1000
        ship       0.79      0.90      0.84      1000
       truck       0.83      0.86      0.85      1000

    accuracy                           0.75     10000
   macro avg       0.75      0.75      0.74     10000
weighted avg       0.75      0.75      0.74     10000
```

12. Visualize Some Predictions

This may give some insight into why the network is misclassifying certain objects.

In [43]:
```
# get predictions on the test set
y_hat = model.predict(x_test)

cifar10_labels = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'tru
ck']
```

In [44]:
```
# plot a random sample of test images, their predicted labels, and ground truth
fig = plt.figure(figsize=(20, 8))
for i, idx in enumerate(np.random.choice(x_test.shape[0], size=36, replace=False)):
    ax = fig.add_subplot(4, 9, i + 1, xticks=[], yticks=[])
    ax.imshow(np.squeeze(x_test[idx]))
    pred_idx = np.argmax(y_hat[idx])
    true_idx = np.argmax(y_test[idx])
    ax.set_title("{} ({})".format(cifar10_labels[pred_idx], cifar10_labels[true_idx]),
                 color=("green" if pred_idx == true_idx else "red"))
```



In [44]:
```
```