## Project-2: Image recognition using CNN

In this module, we will a small dataset, CIFAR-10 which can be downloaded from **Google drive** along with its [labels](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012782825259556864334/web-hosted/assets/cifar10Labels.csv). This dataset consists of 60,000 images of shape 32x32x3 each categorized into one of the 10 categories (airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks). Further, each class has 6000 images.

Convolution Neural Network is similar to multi-layer perceptron having made up of neurons with learnable parameters computing the loss function in the last layer. However, CNN architecture primarily made an explicit assumption that inputs are images but in recent times their applications are seen in the areas of text, speech and time series forecasting.

Let's understand the performance difference between the two using a small example. Consider an image with a dimension of 16x16x3.

# Implementation using Multi-Layer Perceptron

For a classical neural network, all joined connections in its first hidden layer will result in 768 weights (16\*16\*3). However, with an increased number of neurons and image size (say 300x300x3 with 270000 weights) the structure with a vast number of parameters quickly leads to overfitting. Also, MLP are not known to preserve the spatial features within the images.

# Implementation using Convolutional Neural Network

ConvNet pre-assumes the input to be images and hence arranges its neurons in a 3D volume structure: width, height, and depth.

So, an image of shape 16x16x3 will form a volume of similar shape i.e. 16x16x3 with neurons connecting only to a slice of preceding layer neurons. The resulting output has a dimension of 1x1xh where h represents the target labels.

Let us learn what are the components of a CNN and how does it work.

A basic CNN architecture consists of the following layers:

1. Input -> Convolutional -> Pooling -> Output

Among the Convolutional and Pooling layers, both can be repeated as many times as you like.

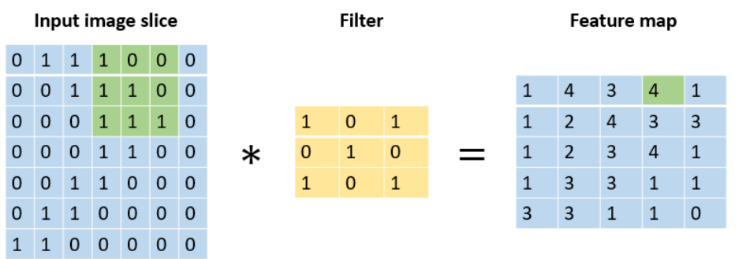
# Input layer

Input layer having two dimensions works as a storage unit for holding raw image data with the preferable size in a multiple of 16, 32, 64, 224, 256, etc. for both height and width for the efficient use of memory fields.

# Convolutional Layer

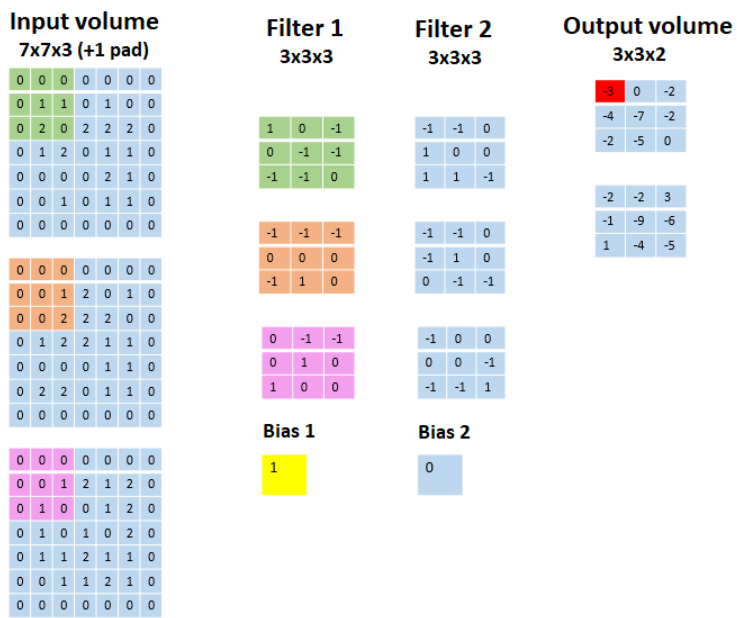
Once the image is loaded in the input layer, the succeeding hidden layers connect back to their preceding layers only on a local region known as the receptive field. This follows a convolution operation which is a combined integration between two functions. It depicts how one function modifies the shape of others.

Since images are represented as a form of a multi-dimensional matrix in the system, therefore, consider the below picture to learn how convolution takes place on a channel (RGB) of an image:

`Here, at a time a certain image slice is chosen. The filter slides over the input volume convolving with the local region at a time. The number of pixels to jump for next convolution is governed by the stride. Stride with value 1 makes the filter slide over the input volume with 1 pixel, a value of 2 makes the filter slide with 2 pixels and so on. Larger the stride, lesser the spatial extent of output volume. In this illustration, the stride is taken as 1. Sometimes, the filter size along with stride value doesn't fit the shape of the image, therefore, in such cases, extra padding of zero is preferred. Zero-padding across the input volume border provides us two benefits: first, it helps retain the border information of the image. Since with each convolution, the size of the image keeps reducing and hence without padding the border information may simply be removed. Second, it helps keep the shape of input and output volume equal. Since filter convolution may change the output volume spatial extent, hence padding helps to avoid such cases.

During the process, you can choose the number of filters where each one of them locates distinct features likes edges, blobs, etc. Distinct filters are indeed necessary to gain distinct features in the process. For instance, with distinct filters, we can attain features including a sharp image, blurred image, image edges, etc.

To wrap up this idea in one single example, consider the given image where we choose a stride value of two, zero-padding value as one along with two filters. So, to arrive at the output -3 (colored in red), you need to get the sum of the pointwise multiplication of the similar colored matrixes.

For instance,

Output volumeRed = Input volumeGreen \* Filter 1Green + Input volumeOrange \* Filter 1Orange + Input volumePink \* Filter 1Pink + Bias 1

Similarly, you can proceed to find the values of other cells of the output matrixes. Note, the first output volume matrix is formed using Filter 1 and Bias 1 whereas the second output volume matrix is formed by Filter 2 and Bias 2.

# Pooling layer

This layer performs downsampling operation along the two dimensions (width and height), hence reducing the number of required parameters and thus reduced computation and a lesser chance of overfitting. It uses the MAX function and requires two hyperparameters the receptive field, and the stride rate. Padding is generally not used with pooling layer. Also, it doesn’t introduce any new parameter as it works on a fixed function.



**An alternative to pooling layer:** Jost Tobias Springenberg et.al. in their paper "Striving for Simplicity: The All Convolutional Net" suggests that using a higher stride once and using only CONV layers can completely remove the need of having a pooling layer in the architecture.

# Fully-Connected layer

Neurons in the Fully-Connected layer are connected to all the activations in previous layers as in ordinary neural networks. It uses the softmax activation function for classifying input images into various classes.

This page lists the conventions required in the process:

If you input a volume of size W1 \* H1 \* D1, it requires four hyperparameters:

1. Number of filters, K
2. Receptive field, F
3. The stride, S
4. The amount of zero-padding, P

Which produces a volume of size W2 \* H2 \* D2 where

* W2=(W1-F+2P)/(S+1)
* H2=(H1-F+2P)/(S+1)
* D2=K

**CNN implementation from scratch: Overview**

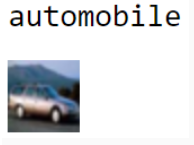
In the previous resource, you've learned the basics of CNN. In this resource, you'll learn to code CNN from scratch using CIFAR-10 dataset by having hands on its hyperparameters, visualization of each layer and much more.

Let us start by importing basic modules:

1. from matplotlib import pyplot as plt
2. %matplotlib inline
3. from sklearn.preprocessing import LabelEncoder
4. import keras
5. import pandas as pd
6. import numpy as np
7. from PIL import Image
8. import os
9. import warnings
10. warnings.filterwarnings('ignore')

Next, let us import the label file and view any random image along with its label:

1. labels = pd.read\_csv('cifar10\_Labels.csv', index\_col=0)
2. *# View an image*
3. img\_idx = 5
4. print(labels.label[img\_idx])
5. Image.open('cifar10/'+str(img\_idx)+'.png')



As we can observe the label is correct as per the image. Now, let us split the data into training and test, follow up with its transformation and normalization:

1. *# Splitting data into Train and Test data*
2. from sklearn.model\_selection import train\_test\_split
3. y\_train, y\_test = train\_test\_split(labels.label, test\_size=0.3, random\_state=42)
4. train\_idx, test\_idx = y\_train.index, y\_test.index *# Stroing indexes for later use*
5. *# Reading images for training*
6. temp = []
7. for img\_idx in y\_train.index:
8. img\_path = os.path.join('cifar10/', str(img\_idx) + '.png')
9. img = np.array(Image.open(img\_path)).astype('float32')
10. temp.append(img)
11. X\_train = np.stack(temp)
12. *# Reading images for testing*
13. temp = []
14. for img\_idx in y\_test.index:
15. img\_path = os.path.join('cifar10/', str(img\_idx) + '.png')
16. img = np.array(Image.open(img\_path)).astype('float32')
17. temp.append(img)
18. X\_test = np.stack(temp)
19. *# Normalizing image data*
20. X\_train = X\_train/255.
21. X\_test = X\_test/255.

The next preprocessing step it to label encode the image respective labels:

1. *# One-hot encoding 10 output classes*
2. encode\_X = LabelEncoder()
3. encode\_X\_fit = encode\_X.fit\_transform(y\_train)
4. y\_train = keras.utils.np\_utils.to\_categorical(encode\_X\_fit)

Now, let us define the CNN network:

1. *# Defining CNN network*
2. num\_classes = 10
3. model = keras.models.Sequential([
4. *# Adding first convolutional layer*
5. keras.layers.Conv2D(filters=32, kernel\_size=(3, 3), strides=1, padding='same', activation='relu',
6. kernel\_regularizer=keras.regularizers.l2(0.001), input\_shape=(32, 32, 3), name='Conv\_1'),
7. *# Normalizing the parameters from last layer to speed up the performance (optional)*
8. keras.layers.BatchNormalization(name='BN\_1'),
9. *# Adding first pooling layer*
10. keras.layers.MaxPool2D(pool\_size=(2, 2), name='MaxPool\_1'),
11. *# Adding second convolutional layer*
12. keras.layers.Conv2D(filters=64, kernel\_size=(3, 3), strides=1, padding='same', activation='relu',
13. kernel\_regularizer=keras.regularizers.l2(0.001), name='Conv\_2'),
14. keras.layers.BatchNormalization(name='BN\_2'),
15. *# Adding second pooling layer*
16. keras.layers.MaxPool2D(pool\_size=(2, 2), name='MaxPool\_2'),
17. *# Flattens the input*
18. keras.layers.Flatten(name='Flat'),
19. *# Fully-Connected layer*
20. keras.layers.Dense(num\_classes, activation='softmax', name='pred\_layer')
21. ])

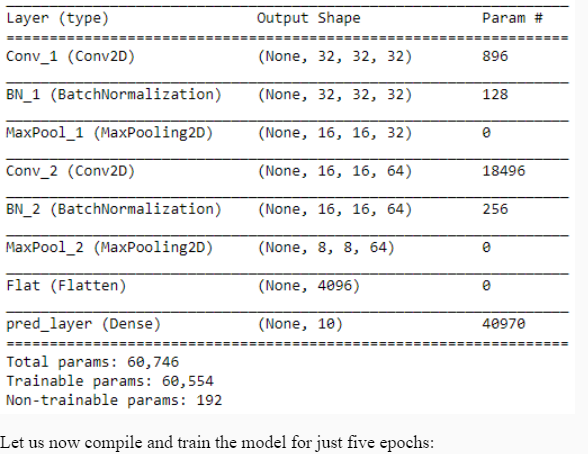
In the above model, we have used two convolution layers paired with max pool layers finally connecting with the Fully-Connected layer. We kept the "same" padding i.e., the output volume will have the same length as the original input. For no padding, you can choose the 'valid' argument. The stride is chosen as 1, a total number of 32 and 64 filters for each respective convolution layer and lastly keeping the kernel size as 3x3.

L2 regularization has been added to cost function via the convolution layers. Also, there's an addition of a new concept termed Batch Normalization. It is added due to the following reasons:

* Since we normalize the data before passing it to the input layer to increase the performance, therefore, we add this extra layer of normalization to normalize the values at the intermediate steps.
* It doesn't let the activation go higher or lower, therefore, you can use a higher learning rate to check for new feature possibilities.
* It works in complement to the dropout regularization. It has a slight regularization property as it adds some noise to each hidden layer activations and thus helps avoid overfitting.

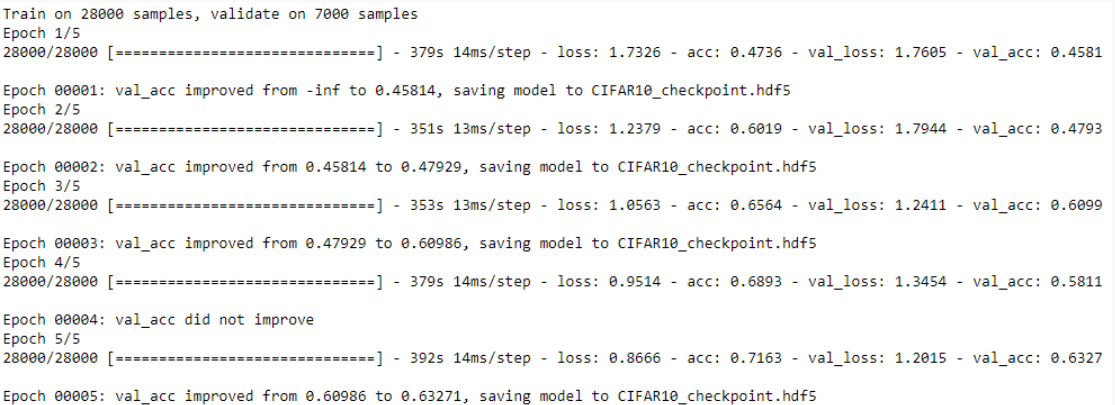
Given below is the summary of the above network:

1. model.summary()



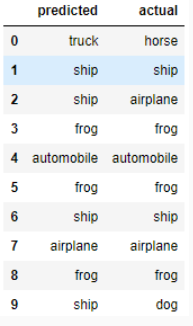
Let us now compile and train the model for just five epochs:

1. *# Compiling the model*
2. model.compile(loss='categorical\_crossentropy',
3. optimizer=keras.optimizers.Adam(),
4. metrics=['accuracy'])
5. cpfile = r'CIFAR10\_checkpoint.hdf5' *# Weights to be stored in HDF5 format*
6. cb\_checkpoint = keras.callbacks.ModelCheckpoint(cpfile, monitor='val\_acc', verbose=1, save\_best\_only=True, mode='max')
7. epochs = 5
8. model.fit(X\_train, y\_train, epochs=epochs, validation\_split=0.2, callbacks=[cb\_checkpoint])



In every batch, the validation accuracy and training accuracy differs much showing a sign of overfitting. However, this model is just to provide you a basic instinct of developing a CNN model from scratch. You can further tune its hyperparameters to increase the performance.

Now, with the given model, let us now perform prediction:

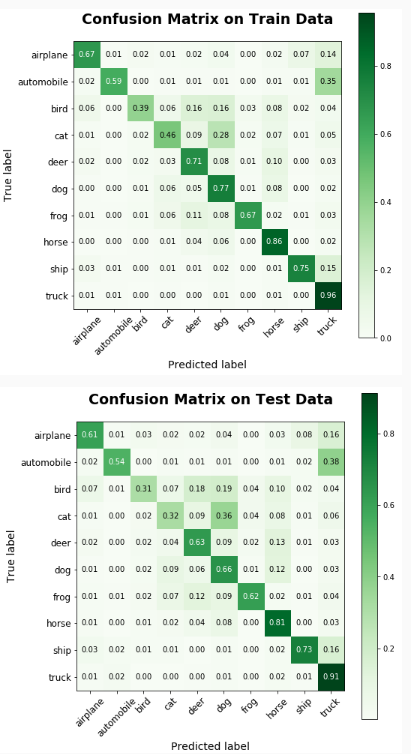
1. *# << DeprecationWarning: The truth value of an empty array is ambiguous >> can arise due to a NumPy version higher than 1.13.3.*
2. *# The issue will be updated in upcoming version.*
3. pred = encode\_X.inverse\_transform(model.predict\_classes(X\_test[:10]))
4. act = y\_test[:10]
5. res = pd.DataFrame([pred, act]).T
6. res.columns = ['predicted', 'actual']
7. res

We can further proceed with train and test accuracy along with the confusion matrix to judge which class the model is predicting better:

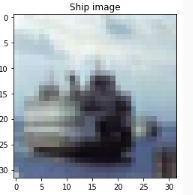
1. from mlxtend.evaluate import scoring
2. train\_acc = scoring(encode\_X.inverse\_transform(model.predict\_classes(X\_train)),
3. encode\_X.inverse\_transform([np.argmax(x) for x in y\_train]))
4. test\_acc = scoring(encode\_X.inverse\_transform(model.predict\_classes(X\_test)), y\_test)
5. print('Train accuracy: ', np.round(train\_acc, 5))
6. print('Test accuracy: ', np.round(test\_acc, 5))

Train Accuracy: 0.3176 & Test Accuracy:0.3874

1. from mlxtend.evaluate import confusion\_matrix
2. from mlxtend.plotting import plot\_confusion\_matrix
3. def plot\_cm(cm, text):
4. class\_names=['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
5. plot\_confusion\_matrix(conf\_mat=cm,
6. colorbar=True, figsize=(8, 8), cmap='Greens',
7. show\_absolute=False, show\_normed=True)
8. tick\_marks = np.arange(len(class\_names))
9. plt.xticks(tick\_marks, class\_names, rotation=45, fontsize=12)
10. plt.yticks(tick\_marks, class\_names, fontsize=12)
11. plt.xlabel('Predicted label', fontsize=14)
12. plt.ylabel('True label', fontsize=14)
13. plt.title(text, fontsize=19, weight='bold')
14. plt.show()
15. *# Train Accuracy*
16. train\_cm = confusion\_matrix(y\_target=encode\_X.inverse\_transform([np.argmax(x) for x in y\_train]),
17. y\_predicted=encode\_X.inverse\_transform(model.predict\_classes(X\_train)),
18. binary=False)
19. plot\_cm(train\_cm, 'Confusion Matrix on Train Data')
20. *# Test Accuracy*
21. test\_cm = confusion\_matrix(y\_target=y\_test,
22. y\_predicted=encode\_X.inverse\_transform(model.predict\_classes(X\_test)),
23. binary=False)
24. plot\_cm(test\_cm, 'Confusion Matrix on Test Data')

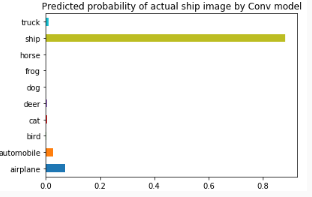
The given model has quite a low accuracy on both train and test data, yet its prediction on categories like a truck, horse, and ship is quite remarkable.

Now, let us learn to see what the dense layer of a model visualizes. For this, we will take our previous model and perform a prediction as shown:

1. from vis.visualization import visualize\_saliency, visualize\_cam, overlay
2. from vis.utils import utils
3. *# Indexes of categories for our model*
4. classes = encode\_y.inverse\_transform(np.arange(10))
5. classes
6. *# array(['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog',*
7. *# 'horse', 'ship', 'truck'], dtype=object)*
8. *# Fetching the ship image*
9. ship\_img = utils.load\_img('cifar10/'+str(test\_idx[6])+'.png') *# can use Image.open() also.*
10. plt.imshow(ship\_img)
11. plt.title('Ship image')
12. plt.show()

The current image is zoomed for illustration purposes.

Since we know that given target is a ship, now let us predict the probability of the classes based on our model.

1. *# Predicting the probability for each of the class*
2. ship\_prob = model.predict(X\_test[6:7]).ravel().copy()
3. pd.Series(ship\_prob, index=classes).plot.barh()
4. plt.title('Predicted probability of actual ship image by Conv model')
5. plt.show()

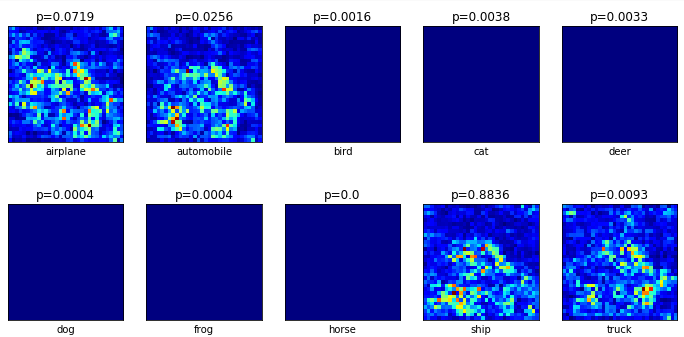
So, the model has presented a high probability that the given image belongs to the ship class.

To visualize activation over final dense layer outputs, we need to switch the softmax activation to linear since the gradient of the output node will depend on all the other node activations.

1. *# Utility to search for layer index by name.*
2. layer\_idx = utils.find\_layer\_idx(model, 'pred\_layer')
3. *# Swap softmax with linear*
4. model.layers[layer\_idx].activation = keras.activations.linear
5. model = utils.apply\_modifications(model)

# 1. Saliency map

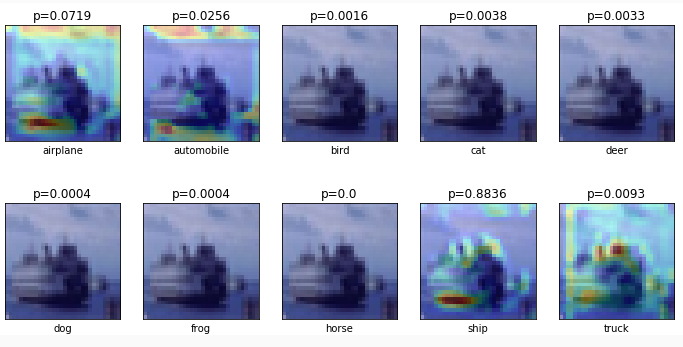
Saliency maps clarify which part does our model focuses on to get a prediction.

1. plt.figure(figsize=(12,6))
2. for i in range(len(classes)):
3. plt.subplot(2, 5, i + 1)
4. grads = visualize\_saliency(model, layer\_idx, filter\_indices=i, seed\_input=ship\_img, backprop\_modifier='guided')
5. plt.xticks([])
6. plt.yticks([])
7. plt.xlabel(classes[i])
8. plt.title('p=' + str(np.round(ship\_prob[i], 4)))
9. plt.imshow(grads, cmap='jet')
10. plt.show()

As shown in the probability graph only the first two and last two nodes are active, leaving center six almost dead. The second last node meant for the ship has the highest probability and as seen in the above plot only emphasize on the region of the ship. Other three active nodes include noise and thus reduced the probability.

# 2. Class activation maps or Grad-cam

These maps contain more detail since they use Conv or Pooling features that contain more spatial detail which is lost in Dense layers. The only additional detail compared to saliency is the penultimate\_layer\_idx. This specifies the pre-layer whose gradients should be used.

1. plt.figure(figsize=(12,6))
2. for i in range(len(classes)):
3. plt.subplot(2, 5, i + 1)
4. cam\_grads = visualize\_cam(model, layer\_idx, filter\_indices=i, seed\_input=ship\_img, backprop\_modifier='guided',
5. penultimate\_layer\_idx=utils.find\_layer\_idx(model, 'BN\_2'))# batch\_normalization\_14
6. plt.xticks([])
7. plt.yticks([])
8. plt.xlabel(classes[i])
9. plt.title('p=' + str(np.round(ship\_prob[i], 4)))
10. plt.imshow(overlay(cam\_grads, ship\_img, alpha=0.3))
11. plt.show()

The ninth subplot constitutes of a contour plot overlayed over the original image which captures ship structure better than other three contour plots which are heavily affected by noise. Center six nodes are almost dead and thus no contour plot.

**CNN hyperparameter tuning**

In the previous resource, we have built a CNN model with stride 1, and filter size as (3x3). In this resource, let us build four CNN models by varying these two parameters and checking how does the accuracy is being affected.

# Case 1 - Stride: 2 and Filter size: (3x3)

1. num\_classes = 10
2. model = keras.models.Sequential([
3. *# Adding first convolutional layer*
4. keras.layers.Conv2D(filters=32, kernel\_size=(3, 3), strides=2, padding='same', activation='relu',
5. kernel\_regularizer=keras.regularizers.l2(0.001), input\_shape=(32, 32, 3), name='Conv\_1'),
6. *# Normalizing the parameters from last layer to speed up the performance (optional)*
7. keras.layers.BatchNormalization(name='BN\_1'),
8. *# Adding first pooling layer*
9. keras.layers.MaxPool2D(pool\_size=(2, 2), name='MaxPool\_1'),
10. *# Adding second convolutional layer*
11. keras.layers.Conv2D(filters=64, kernel\_size=(3, 3), strides=2, padding='same', activation='relu',
12. kernel\_regularizer=keras.regularizers.l2(0.001), name='Conv\_2'),
13. keras.layers.BatchNormalization(name='BN\_2'),
14. *# Adding second pooling layer*
15. keras.layers.MaxPool2D(pool\_size=(2, 2), name='MaxPool\_2'),
16. *# Flattens the input*
17. keras.layers.Flatten(name='Flat'),
18. *# Fully-Connected layer*
19. keras.layers.Dense(num\_classes, activation='softmax', name='pred\_layer')
20. ])
21. model.compile(loss='categorical\_crossentropy',
22. optimizer=keras.optimizers.Adam(),
23. metrics=['accuracy'])
24. cpfile = r'CIFAR10\_checkpoint\_stride\_2.hdf5' *# Weights to be stored in HDF5 format*
25. cb\_checkpoint = keras.callbacks.ModelCheckpoint(cpfile, monitor='val\_acc', verbose=1, save\_best\_only=True, mode='max')
26. epochs = 5
27. model.fit(X\_train, y\_train, epochs=epochs, validation\_split=0.2, callbacks=[cb\_checkpoint])
28. from mlxtend.evaluate import scoring
29. actual\_train = [np.argmax(x) for x in y\_train]
30. predicted\_train = model.predict\_classes(X\_train)
31. print('Train accuracy: ', scoring(actual\_train, predicted\_train, metric='accuracy') \* 100) *# Change metric for more.*
32. actual\_test = [np.argmax(x) for x in y\_test]
33. predicted\_test = model.predict\_classes(X\_test)
34. print('Test accuracy: ', scoring(actual\_test, predicted\_test, metric='accuracy') \* 100) *# Change metric for more.*

**Train Accuracy: 50.74857142857143 & Test Accuracy: 47.893333**

# Case 2 - Stride: 3 and Filter size: (3x3)

1. num\_classes = 10
2. model = keras.models.Sequential([
3. *# Adding first convolutional layer*
4. keras.layers.Conv2D(filters=32, kernel\_size=(3, 3), strides=3, padding='same', activation='relu',
5. kernel\_regularizer=keras.regularizers.l2(0.001), input\_shape=(32, 32, 3), name='Conv\_1'),
6. *# Normalizing the parameters from last layer to speed up the performance (optional)*
7. keras.layers.BatchNormalization(name='BN\_1'),
8. *# Adding first pooling layer*
9. keras.layers.MaxPool2D(pool\_size=(2, 2), name='MaxPool\_1'),
10. *# Adding second convolutional layer*
11. keras.layers.Conv2D(filters=64, kernel\_size=(3, 3), strides=3, padding='same', activation='relu',
12. kernel\_regularizer=keras.regularizers.l2(0.001), name='Conv\_2'),
13. keras.layers.BatchNormalization(name='BN\_2'),
14. *# Adding second pooling layer*
15. keras.layers.MaxPool2D(pool\_size=(2, 2), name='MaxPool\_2'),
16. *# Flattens the input*
17. keras.layers.Flatten(name='Flat'),
18. *# Fully-Connected layer*
19. keras.layers.Dense(num\_classes, activation='softmax', name='pred\_layer')
20. ])
21. model.compile(loss='categorical\_crossentropy',
22. optimizer=keras.optimizers.Adam(),
23. metrics=['accuracy'])
24. cpfile = r'CIFAR10\_checkpoint\_stride\_3.hdf5' *# Weights to be stored in HDF5 format*
25. cb\_checkpoint = keras.callbacks.ModelCheckpoint(cpfile, monitor='val\_acc', verbose=1, save\_best\_only=True, mode='max')
26. epochs = 5
27. model.fit(X\_train, y\_train, epochs=epochs, validation\_split=0.2, callbacks=[cb\_checkpoint])
28. actual\_train = [np.argmax(x) for x in y\_train]
29. predicted\_train = model.predict\_classes(X\_train)
30. print('Train accuracy: ', scoring(actual\_train, predicted\_train, metric='accuracy') \* 100) *# Change metric for more.*
31. actual\_test = [np.argmax(x) for x in y\_test]
32. predicted\_test = model.predict\_classes(X\_test)
33. print('Test accuracy: ', scoring(actual\_test, predicted\_test, metric='accuracy') \* 100) *# Change metric for more.*

**Train Accuracy: 54.58 & Test Accuracy: 50.78666666**

# Case 3 - Stride 1 and Filter size: (2x2)

1. num\_classes = 10
2. model = keras.models.Sequential([
3. *# Adding first convolutional layer*
4. keras.layers.Conv2D(filters=32, kernel\_size=(2, 2), strides=1, padding='same', activation='relu',
5. kernel\_regularizer=keras.regularizers.l2(0.001), input\_shape=(32, 32, 3), name='Conv\_1'),
6. *# Normalizing the parameters from last layer to speed up the performance (optional)*
7. keras.layers.BatchNormalization(name='BN\_1'),
8. *# Adding first pooling layer*
9. keras.layers.MaxPool2D(pool\_size=(2, 2), name='MaxPool\_1'),
10. *# Adding second convolutional layer*
11. keras.layers.Conv2D(filters=64, kernel\_size=(2, 2), strides=1, padding='same', activation='relu',
12. kernel\_regularizer=keras.regularizers.l2(0.001), name='Conv\_2'),
13. keras.layers.BatchNormalization(name='BN\_2'),
14. *# Adding second pooling layer*
15. keras.layers.MaxPool2D(pool\_size=(2, 2), name='MaxPool\_2'),
16. *# Flattens the input*
17. keras.layers.Flatten(name='Flat'),
18. *# Fully-Connected layer*
19. keras.layers.Dense(num\_classes, activation='softmax', name='pred\_layer')
20. ])
21. model.compile(loss='categorical\_crossentropy',
22. optimizer=keras.optimizers.Adam(),
23. metrics=['accuracy'])
24. cpfile = r'CIFAR10\_checkpoint\_filter\_2.hdf5' *# Weights to be stored in HDF5 format*
25. cb\_checkpoint = keras.callbacks.ModelCheckpoint(cpfile, monitor='val\_acc', verbose=1, save\_best\_only=True, mode='max')
26. epochs = 5
27. model.fit(X\_train, y\_train, epochs=epochs, validation\_split=0.2, callbacks=[cb\_checkpoint])
28. actual\_train = [np.argmax(x) for x in y\_train]
29. predicted\_train = model.predict\_classes(X\_train)
30. print('Train accuracy: ', scoring(actual\_train, predicted\_train, metric='accuracy') \* 100) *# Change metric for more.*
31. actual\_test = [np.argmax(x) for x in y\_test]
32. predicted\_test = model.predict\_classes(X\_test)
33. print('Test accuracy: ', scoring(actual\_test, predicted\_test, metric='accuracy') \* 100) *# Change metric for more.*

**Train Accuracy: 69.79714285714286 & Test Accuracy: 62.706666**

# Case 4 - Stride: 1 and Filter size: (4x4)

1. num\_classes = 10
2. model = keras.models.Sequential([
3. *# Adding first convolutional layer*
4. keras.layers.Conv2D(filters=32, kernel\_size=(4, 4), strides=1, padding='same', activation='relu',
5. kernel\_regularizer=keras.regularizers.l2(0.001), input\_shape=(32, 32, 3), name='Conv\_1'),
6. *# Normalizing the parameters from last layer to speed up the performance (optional)*
7. keras.layers.BatchNormalization(name='BN\_1'),
8. *# Adding first pooling layer*
9. keras.layers.MaxPool2D(pool\_size=(2, 2), name='MaxPool\_1'),
10. *# Adding second convolutional layer*
11. keras.layers.Conv2D(filters=64, kernel\_size=(4, 4), strides=1, padding='same', activation='relu',
12. kernel\_regularizer=keras.regularizers.l2(0.001), name='Conv\_2'),
13. keras.layers.BatchNormalization(name='BN\_2'),
14. *# Adding second pooling layer*
15. keras.layers.MaxPool2D(pool\_size=(2, 2), name='MaxPool\_2'),
16. *# Flattens the input*
17. keras.layers.Flatten(name='Flat'),
18. *# Fully-Connected layer*
19. keras.layers.Dense(num\_classes, activation='softmax', name='pred\_layer')
20. ])
21. model.compile(loss='categorical\_crossentropy',
22. optimizer=keras.optimizers.Adam(),
23. metrics=['accuracy'])
24. cpfile = r'CIFAR10\_checkpoint\_filter\_4.hdf5' *# Weights to be stored in HDF5 format*
25. cb\_checkpoint = keras.callbacks.ModelCheckpoint(cpfile, monitor='val\_acc', verbose=1, save\_best\_only=True, mode='max')
26. epochs = 5
27. model.fit(X\_train, y\_train, epochs=epochs, validation\_split=0.2, callbacks=[cb\_checkpoint])
28. actual\_train = [np.argmax(x) for x in y\_train]
29. predicted\_train = model.predict\_classes(X\_train)
30. print('Train accuracy: ', scoring(actual\_train, predicted\_train, metric='accuracy') \* 100) *# Change metric for more.*
31. actual\_test = [np.argmax(x) for x in y\_test]
32. predicted\_test = model.predict\_classes(X\_test)
33. print('Test accuracy: ', scoring(actual\_test, predicted\_test, metric='accuracy') \* 100) *# Change metric for more.*

**Train Accuracy: 71.342857142857714 & Test Accuracy: 64.246667**

# **Conclusion**

Here's the table which compares the performance of the above four models with the basic model we built earlier:

As you can observe from the above table, just by varying two hyperparameters the accuracy changes dramatically. Even with very large filter size and stride rate, the model may not be able to learn small spatial features.